HOCHSCHULE
MITTWEIDA
UNIVERSITY OF
APPLIED SCIENCES

BACHELORARBEIT

Herr
Eric Schröder

Konzeptionierung eines embedded Device zur Aufzeichnung von Datenverkehr in Drahtlosnetzwerken

2017

Fakultät **Angewandte Computer- und Biowissenschaften**

BACHELORARBEIT

Konzeptionierung eines embedded Device zur Aufzeichnung von Datenverkehr in Drahtlosnetzwerken

Autor:
Eric Schröder

Studiengang:
Allgemeine und Digitale Forensik

Seminargruppe:
FO14W2-B

Erstprüfer:
Prof. Dr. rer. nat. Christian Hummert

Zweitprüfer:
Prof. Dr. rer. pol. Dirk Pawlaszczyk

Mittweida, September 2017

Faculty of **Applied Computer Sciences & Biosciences**

BACHELOR THESIS

Conception of an embedded device for capturing data traffic in wireless networks

Author:
Eric Schröder

Study Programme:
General and Digital Forensic Science

Seminar Group:
FO14W2-B

First Referee:
Prof. Dr. rer. nat. Christian Hummert

Second Referee:
Prof. Dr. rer. pol. Dirk Pawlaszczyk

Mittweida, September 2017

Bibliografische Angaben

Schröder, Eric: Konzeptionierung eines embedded Device zur Aufzeichnung von Datenverkehr in Drahtlosnetzwerken, 81 Seiten, 15 Abbildungen, Hochschule Mittweida, University of Applied Sciences, Fakultät Angewandte Computer- und Biowissenschaften

Bachelorarbeit, 2017

Referat

In der vorliegenden Bachelorarbeit wird eine Methodik vorgestellt, den Datenverkehr in Drahtlosnetzwerken mithilfe eines embedded Device aufzuzeichnen. Zu herkömmlichen, kommerziellen Produkten wird hier, basierend auf den von „Espressif“ entwickelten ESP32, eine kostengünstige Alternative dargestellt. Der aufgezeichnete Netzwerkverkehr kann innerhalb des Programmablaufs gefiltert, sowie anschließend auf einer SD-Karte gesichert werden. Der in dieser Arbeit aufgezeigte Schaltplan ist, wie auch der Programmablauf, auf den jeweiligen Einsatz modifizierbar. Aufgrund der geringen Größe von etwa 52mm x 27mm x 13mm ist das embedded Device für den mobilen Einsatz prädestiniert. Die Performance des Geräts und der verwendeten SD-Karte werden jeweils in einem Benchmark veranschaulicht. Aufgrund der Geschwindigkeit von 0,005284 Mbps kann ein lückenloses Aufzeichnen von drahtlosen Netzwerkverkehr zwar nicht gewährleistet, jedoch im Ausblick alternative Einsatzgebiete vorgeschlagen werden.

Bibliographic Information

Schröder, Eric: Conception of an embedded device for capturing data traffic in wireless networks, 81 pages, 15 figures, Hochschule Mittweida, University of Applied Sciences, Faculty of Applied Computer Sciences & Biosciences

bachelor thesis, 2017

Abstract

This bachelor thesis presents a methodology for capturing data traffic in wireless networks using an embedded device. This is based on the ESP32 developed by „Espressif“ and thus represents a cost-effective alternative to currently commercially available products. The captured network traffic can be filtered within the program sequence and is then saved on a SD card. The circuit diagram can be taken from this work and, like the program sequence, can be modified for the particular application. Due to the small size of approximately 52mm x 27mm x 13mm, the embedded device is predestined for mobile use. The performance of the device and the SD card used are shown in a benchmark. Due to the speed of 0.005284 Mbps, seamless recording of wireless network traffic can not be ensured, but alternative scenarios are proposed in outlook.

I. Inhaltsverzeichnis

II. Abbildungsverzeichnis

III. Tabellenverzeichnis

IV. Abkürzungsverzeichnis

AES	Advanced Encryption Standard
API	application programming interface
App	Application software
ARM	Advanced RISC Machines
ARP	Address Resolution Protocol
ATCA	Average Time Channel Allucation
bps	byte per second
BSS	Basic Service Set
BSSID	Basic Service Set ID
CACFA	Channel Activeness Based Channel Fair Allocation
CAOCA	Channel Activeness Based Optimized Channel Allocation
CCMP	Counter Mode with Cipher Block Chaining Message Authentication Code Protocol
CPU	central processing unit
DA	Destination Address
dBm	Dezibel Milliwatt
DSP	Digitaler Signalprozessor
DSSS	Direct Sequence Spread Spectrum
EAP	Extensible Authentication Protocol
ESP-IDF	Espressif IoT Development Framework
ESS	Extended Service Set
ESSID	Extended Service Set Identifier
FAT	File Allocation Table
FCA	Fixed Channel Allocation
FCS	Frame Check Sequence
GHz	Gigahertz
GND	Ground
GPIO	general purpose input/output
GPS	Global Positioning System
GSM	Global System for Mobile Communications
HF	Hochfrequenz
Hz	Hertz

IBSS ..... Independent Basic Service Set
IDE ..... Integrierte Entwicklungsumgebung
IEEE ..... Institutes of Electrical and Electronics Engineers
IRQ ..... Interrupt Request
IT ..... Informationstechnik
KB ..... Kilobyte
kB/s ..... Kilobytes per second
kHz ..... Kilohertz
LED ..... lichtemittierende Diode
Li-Ion ..... Lithium-Ionen
mA ..... Milliampere
MAC ..... Multiply-Accumulate
MB/s ..... Megabytes per second
MB/s ..... Megabytes per second
Mbps ..... Megabits per second
MCU ..... Microcontroller Unit
MHz ..... Megahertz
MIC ..... Message Integrity Check
nF ..... Nanofarad
NTP ..... Network Time Protocol
OSI ..... Open Systems Interconnection Model
OUI ..... Organizationally Unique Identifier
PPM ..... parts per million
QSPI ..... Quad serial peripheral interface
RA ..... Receiver Address
RESTful ..... Representational State Transfer
ROM ..... read-only memory
RSN ..... Robust Security Network
RTC ..... real-time clock
RTOS ..... real-time operating system
RX ..... Receiver exchange
SA ..... Source Address
SD ..... Secure Digital
SPI ..... serial peripheral interface
SRAM ..... random-access memory

SSID ..... Service Set Identifier

TA ..... Transmitter

TKIP ..... Temporal Key Integrity Protocol

TX ..... Transmitter exchange

UART ..... Universal Asynchronous Receiver Transmitter

ULP ..... Ultra-Low-Voltage-Prozessor

USB ..... Universal Serial Bus

WEP ..... Wired Equivalent Privacy

WPA ..... Wi-Fi Protected Access

WPA2 ..... Wi-Fi Protected Access 2

1 Einleitung

Die Überwachung von Netzwerken (auch Network-Monitoring genannt) hat im IT-Systemmanagement, sei es für private oder wirtschaftliche Zwecke, einen besonderen Stellenwert. Hierbei handelt es sich um regelmäßige Kontrollen der Kommunikationsverbindungen von Netzwerken, um diese gegen potenzielle Angriffe abzusichern, oder nach (bzw. auch während) eines erfolgten Angriffes digitale Spuren zu sichern. Auch auftretende Protokollfehler und diverse Probleme innerhalb der Netzwerkstruktur können so erfasst und behoben werden. Diese Form der Netzwerkanalyse wird auch „sniffing“ genannt. Von den Network-Monitoring-Programmen sind „Intrusion Detection Systeme“ abzugrenzen. Während Monitoring-Systeme für die Aufzeichnung des Netzwerkverkehrs spezialisiert sind, sind letztere Systeme dazu konzipiert, Angriffe mittels Muster und Filter zu erkennen.

Solche Netzwerk-Analyseprogramme können jedoch nicht nur für das eigene Netzwerk genutzt werden. Durch das Versetzen von Netzwerkgeräten in spezielle Modi oder Man-in-the-Middle-Angriffen, wie beispielsweise dem ARP-Spoofing, also dem Versenden manipulierter ARP-Pakete an einen oder mehrere Netzwerkteilnehmer, sodass der Datenverkehr dieser über den Angreifer läuft, ist es möglich, den Datenverkehr von fremden Netzwerken aufzuzeichnen.
Die Möglichkeit, solche Funktionen und Abläufe mittels Mikrocontrollern und eingebetteten Systemen zu nutzen, ist aufgrund des schnell voranschreitenden, technischen Fortschritts möglich. Der Einsatz von konventionellen und somit physisch größeren sowie kostenintensiveren Computern wird damit von einer günstigeren Methode abgelöst.

Diese Bachelorarbeit setzt sich sowohl mit den Grundlagen für die Konzeptionierung eines solchen embedded Device auseinander als auch mit der Umsetzung diese, auf Grundlage dieser Erkenntnisse.
Mithilfe eines solchen eingebundenen Gerätes soll der Datenverkehr eines drahtlosen Netzwerkes aufgezeichnet und auf einem externen Medium gespeichert werden. Nach erfolgreicher Aufzeichnung wird dieser Traffic in ein für das Netzwerkanalyse-Programm „Wireshark“ interpretierbares Format konvertiert und anschließend ausgewertet. Mittels eines Benchmarks soll außerdem die Geschwindigkeit des embedded Device festgestellt und anschließend mit anderen Studien verglichen werden.

Angesichts der aktuell auf dem Markt beworbenen Produkte zum Aufzeichnen von drahtlosen Netzwerkverkehr, soll diese Methodik eine kostengünstige Alternative vorstellen.

In Kapitel 2 werden die Grundlagen mithilfe einer Begriffserläuterung sowie Vorstellung der Funktionsweise von Mikrocontrollern beschrieben. Darüber hinaus wird der Forschungsgegenstand „ESP32“ vorgestellt und der IEEE 802.11-Standard erläutert. Im Mittelpunkt von Kapitel 3 stehen die Materialien und Methoden mit dem Aufzeigen von Schaltplänen des embedded Device, der Programmablauf sowie das Berücksichtigen des Stromverbrauchs und der möglichen -versorgung. Gegenstand von Kapitel 4 ist die Präsentation der Forschungsergebnisse in Form eines Wireshark-Mitschnitts und eines Benchmarks. Der Benchmark wird jeweils für das embedded Device wie auch für die SD-Karte ausgeführt und soll zeigen ob das Schreiben auf einer SD-Karte schnell genug ist und zum korrekten Verarbeiten der Daten geeignet ist. Im abschließenden Kapitel wird die Arbeit zusammengefasst und kritisch mit ähnlichen Studien verglichen, um das embedded Device in einen wissenschaftlichen Kontext einzureihen.

2 Grundlagen

2.1 Mikrocontroller

2.1.1 Überblick

Mikrocontroller (auch Microcontroller Unit, kurz MCU) zählen zu den anwendungsorientierten Mikroprozessoren. Durch Merkmale wie einem integrierten Speicher (sowohl Flash- als auch Arbeitsspeicher), einem integrierten Schaltkreis und der Fähigkeit Programmcodes zu speichern und auszuführen, stellen sie Ein-Chip-Systeme dar.
Dabei ist die gesamte Hardware, welche zum Ausführen einfacher Programmabläufe gedacht ist, auf einer einzigen Platine untergebracht. Aufgrund ihrer geringen Größe von teilweise wenigen Millimetern, können Mikrocontroller direkt in Systeme eingebunden werden. In diesem Kontext werden Aufgaben wie Steuerung und Überwachung von Datenverarbeitungssystemen übernommen. Solche Systeme werden daher auch embedded Systems bzw. eingebettete Systeme genannt.

Eingebettete Systeme sind aus dem heutigen technischen Umfeld nicht mehr wegzudenken. Essentielle Abläufe, sei es im Privatleben oder in Unternehmen, werden von solchen Systemen gesteuert und koordiniert [Bä10, S. 3]. Im Gegensatz zu reinen Rechnersystemen besitzen Mikrocontroller eine geringere Rechenleistung bei gleichzeitig geringerem Hardwareaufwand, wobei oftmals auch die finanziellen Aspekte (wie z.B. für die Hardware) im Vordergrund stehen. Einsatzgebiete sind beispielsweise Bereiche, in denen nur eine geringe Rechenleistung für die Aufgabenerfüllung benötigt wird. Daher ergibt sich durch den Einsatz von Mikrocontrollern nicht nur ein finanzieller Vorteil, sondern auch eine Platzersparnis im jeweiligen System [GWUW17, S. 420].

2.1.2 Elementarer Aufbau

Die CPU (central processing unit) wertet Signale aus und führt arithmetische Operationen sowie Befehle aus. Der Datenspeicher stellt einen nichtflüchtigen Flashspeicher dar, welcher u.a. den Programmcode und den Bootloader beinhaltet. Durch externe Einheiten kann dieser Speicher entsprechend erweitert werden (siehe Abschnitt 2.2.3 unter Peripheriegeräte und Sensoren). Der Arbeitsspeicher fungiert als Ablage für temporäre Dateien, um so schnell auf Daten zuzugreifen und stellt damit, im Gegensatz zum Datenspeicher, einen flüchtigen Speicher dar. IRQ (Interrupt Request) dient der Verarbeitung von Interrupts (Unterbrechung des Programmiercodes), um andere kritische Vorgänge zu bearbeiten. Als Peripherie werden Geräte bzw. andere zusätzliche Hardware bezeichnet, welche an die Zentraleinheit angeschlossen und betrieben werden [Som12, S. 14-16].

Zähler lösen bei einem Overflow (Überlauf) einen Interrupt aus. Overflows treten auf, wenn das Zählregister eines Zählers den höchsten Stand erreicht und wieder auf Null zurückgesetzt wird. Das Register wird durch den Oszillator inkrementiert oder dekrementiert. Oftmals werden Prescaler (Vorteiler) genutzt um den Overflow hinauszuzögern, dieser teilt den Takt um einen vorgegebenen Faktor. Das folgende Beispiel erläutert anhand eines 4 MHz Oszillators, eines 8-bit Zählers und einem Prescaler mit dem Faktor 1024 die Berechnung der Overflows pro Sekunde:

$$\frac{\left(\frac{\text{Systemtakt in Hz}}{\text{Prescaler}}\right)}{2^{\text{ Bit des Zählers}}} = \frac{\left(\frac{4\,\text{MHz}\ \cdot\ 1.000.000}{1024}\right)}{2^8} = \frac{\left(\frac{4.000.000\,\text{Hz}}{1024}\right)}{256} = \underline{\underline{15,25\ \text{Overflows/Sek}}}$$

Mithilfe des Prescalers konnten die Overflows von 15625 auf 15,25 gesenkt werden (entspricht alle ~0,0655 Sekunden 1 Overflow). Die Zeit t je Overflow lässt sich daher folgendermaßen ableiten:

$$t = \frac{2^{\text{ Bit des Zählers}}\ \cdot\ \text{Prescaler}}{\text{Systemtakt in Hz}}$$

Jedes mal, wenn ein Overflow ausgelöst wird, wird eine Aktion unabhängig vom restlichen Programmiercode ausgeführt [Sch17a].

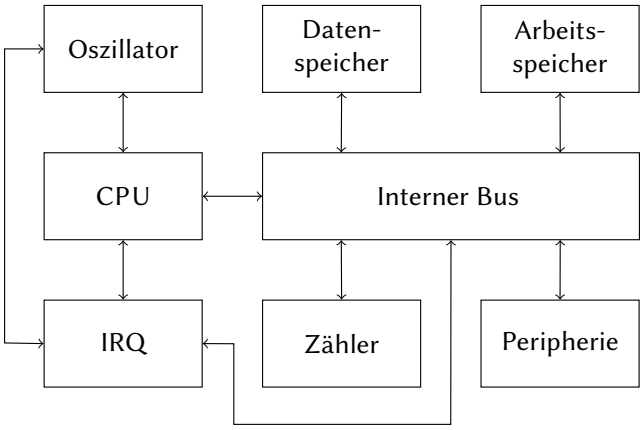


Abbildung 2.1: Aufbau eines Mikrocontrollers [modifiziert nach [Som12, S. 14]]

2.1.3 Abgrenzung gegenüber Einplatinencomputer

Einplatinencomputer stellen vollständige Computersysteme dar, dessen Komponenten wie CPU, GPU, Netzwerkkarte, usw. auf einer einzelnen Platine zusammengefasst sind [Bol17, S. 256]. Im Gegensatz zu Mikrocontrollern besitzen Einplatinencomputer zumeist ein vollständiges Betriebssystem. Einer der berühmtesten Vertreter von Einplatinencomputern ist der im Jahr 2012 erschienene Raspberry Pi. Das am häufigsten be-

nutzte Betriebssystem ist das auf Linux basierende Raspbian (Kunstwort aus Raspberry und Debian[1]). Die Vorteile liegen hier darin, dass es sich bei Linux um kostenfreie Open Source Software handelt. Dadurch kann einerseits der Quelltext von Dritten eingesehen, als auch modifiziert/erweitert werden und andererseits können hohe Lizenzkosten vermieden werden [WR16, S. 5]. In Summe sind die (Software-)Kosten für einen Raspberry Pi somit gering. Neben Raspbian gibt es weitere kostenfreie, auf Linux basierende Betriebssysteme, wie beispielsweise Kali Linux (für Penetrationstests geeignet), LibreELEC (Verwendung des Raspberry Pis als Media Center) oder RTAndroid (Echtzeit Entwicklung von Android-Programmen) . Auch können andere Betriebssysteme wie „Windows 10 IoT Core“ genutzt werden.
Da Einplatinencomputer oftmals über ansprechbare Kontaktpins verfügen, können Projekte für Mikrocontroller ebenso mit Einplatinencomputer realisiert werden. Wie zuvor erwähnt, ist es jedoch notwendig ein entsprechendes Betriebssystem zu installieren. Auf diesen Fakt kann bei Mikrocontrollern verzichtet werden. Die Verwendung eines Betriebssystems verlangt das Hoch- und Runterfahren dessen bei der Verwendung. Ebenfalls ist durch die erhöhte Anzahl an elektronischer Komponenten der Stromverbrauch im Vergleich zu Mikrocontroller um ein Vielfaches höher [Kry15, S.23].

2.2 ESP-WROOM-32

2.2.1 Überblick

Der ESP32 ist ein vom Hersteller „Espressif Systems“ entwickeltes Modul mit integriertem Mikrocontroller, welcher einen 2,4 GHz Wi-Fi und Bluetooth Combo-Chip besitzt und stellt den Nachfolger des ESP8266 dar. Einer der wichtigsten Erneuerungen ist der Dual-Core Mikroprozessor mit einer Taktrate von bis zu 240 MHz [Esp17d, S. 12]. Im Vergleich hierzu besitzt das Vorgänger-Modell einen 80 MHz bis 160 MHz Single-Core Prozessor [Esp17e, S. 7]. Des Weiteren wurde der Promiscuous Mode (siehe Abschnitt 2.3.4.1) als Funktion in der Headerdatei „esp_wifi.h“ deklariert.
Denkbare Einsatzbereiche sind Entwicklungen im Bereich mobiler Anwendungen, tragbarer Elektronik und IoT-Anwendungen [Esp17d, S. 1].

2.2.2 Pin Belegung

Das Development-Board führt je nach Hersteller nahezu alle Kontaktpins des ESP32 Chips nach außen, sodass der Mikrocontroller mit Hilfe eines Steckbretts programmiert werden kann. Wenige Ausnahmen bilden hier beispielsweise der GPIO 0 (in Abbildung 2.2 unten rechts zu erkennen) vom Hersteller DOIT. In solchen Fällen müssen die Pins über einen Jumper zum Steckbrett geführt werden.

[1] Freies Betriebssystem, basierend auf dem Linux-Kernel

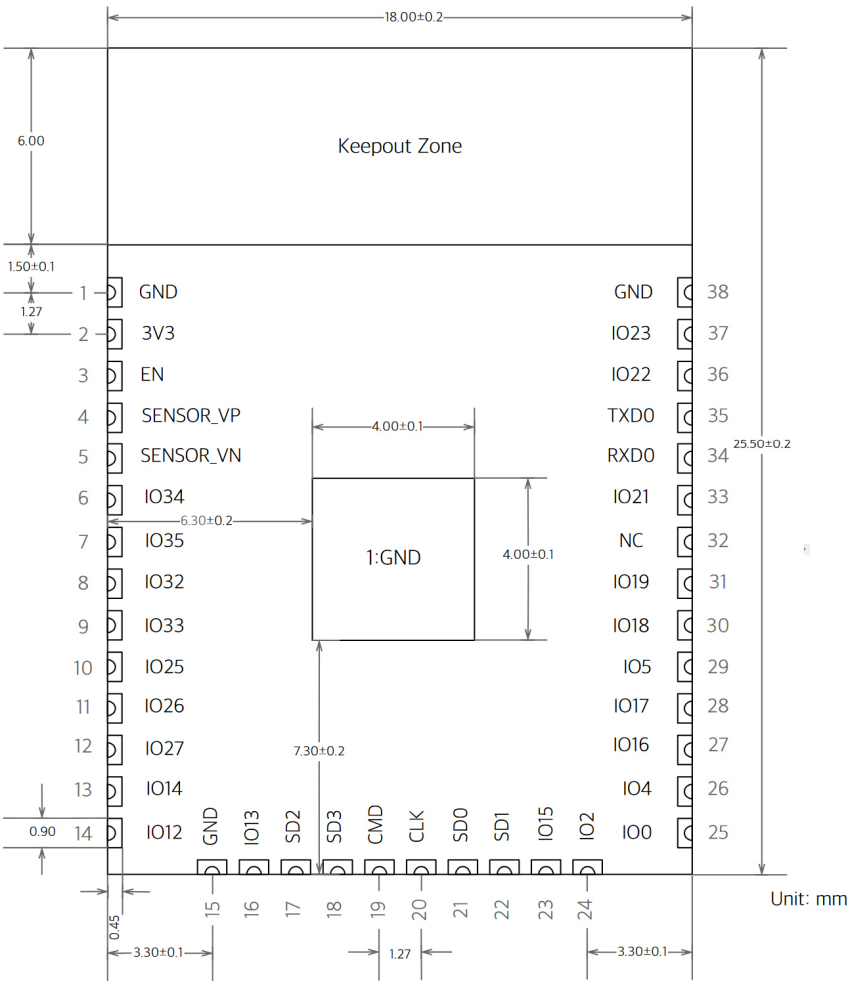


Abbildung 2.2: Pin Layout vom ESP-WROOM-32 [Esp17c, S. 3]

2.2.3 Merkmale und Funktionen

CPU

Der ESP-32 besitzt einen Xtensa Dual-Core 32-bit LX6 Mikroprozessor mit zwei 32-bit CPU-Kernen, welche jeweils eine Taktrate von 80 MHz bis 240 MHz aufweisen [Esp17c, S. 1]. Dieser besitzt eine Gleitkommaeinheit (Koprozessor), welche fähig ist Operationen auf Gleitkommazahlen zu behandeln, um auch rechenintensive Aufgaben auszuführen. Des Weiteren unterstützen die zwei Xtensa-Kerne DSP-Anweisungen wie 32-bit Multiplikationen ($a = a \cdot b$), 32-bit Divisionen ($a = \frac{a}{b}$) und 40-bit MAC-Befehle. ($a \leftarrow a + (b \cdot c)$) [Esp17d, S. 12]. Der DSP dient der digitalen Verarbeitung analoger Signale mittels spezieller Operationen [Bä10, S. 3]. Zur Behandlung von möglichen Ausnahmefällen unterstützen die Mikroprozessoren insgesamt 32 Interruptvektoren von 70 Interruptquellen [Esp17d, S. 12].

Interner Speicher

Für den Boot-Vorgang und für weitere Kernfunktionen, besitzt der ESP-32 als internen

Speicher einen 448 KB ROM. Dies ist ein nicht flüchtiger Festwertspeicher, welcher vom Prozessor nicht beschrieben, sondern nur gelesen werden kann [Bä10, S. 187]. Für Daten und Anweisungen gibt es einen 520 KB SRAM und einen 16 KB SRAM als Real-Time-Clock. Diese 16 KB SRAM sind untergliedert in 8 KB RTC SLOW Memory (welcher vom Koprozessor im Deep-Sleep Modus[2] angesprochen werden kann) und 8 KB FAST Memory (welcher der CPU während des RTC-Boot vom Deep-Sleep Modus zugewiesen ist).
Darüber hinaus ist der ESP-32 zum nachträglichen, dynamischen Programmieren mit einem 1 kbit eFuse ausgestattet, 256 Bits sind dabei für die MAC-Adresse und Chip-Konfiguration reserviert, die restlichen 768 Bits für Flash-Encryption (basierend auf AES zum Schutz des Programmiercodes und der Daten [Esp17d, S. 13]) und Chip-ID [Esp17c, S. 7].

Externer Flashspeicher und SRAM

Der ESP-32 unterstützt bis zu vier externe 16-MB QSPI Flashspeicher und SRAM mit AES-Hardware-Encryption. QSPI ist eine Schnittstelle zur Kommunikation mit externen Flashspeichern durch den SPI Bus [Nor17].

Clocks und Timer

Zur Takterzeugung besitzt der ESP-32 einen internen 8 MHz Oszillator, einen internen RC-Oszillator, einen externen 2 MHz bis 40 MHz Quarzoszillator sowie einen externen 32 kHz, der genau genommen 32.768 kHz mit einer Standardabweichung von $\pm$ 20 PPM [Esp17c, S. 8]) aufweist, Quarzoszillator als Real-Time-Clock [Esp17d, S. 3]. Oszillatoren erzeugen eigenständig periodische Ausgangssignale, deren Verluste aufgrund der Schaltung selbstständig ausgeglichen werden, sodass die Schwingung konstant bleibt. Jedes dieser schwingungsfähigen Systeme besteht aus einem Verstärker (bspw. einem Operationsverstärker) sowie einem frequenzbestimmenden Bauteil, wie RC-Glieder oder Quarz.
Die Form des erzeugten Signals (z.B. Rechtecksignal, Sinussignal, Sägezahnsignal) ist dabei von der Schaltung abhängig. Bei RC-Oszillatoren wird die Frequenz von Widerständen R und Kondensatoren C beeinflusst, während bei Quarz-Oszillatoren die Frequenz von der harmonischen Schwingung eines Schwingquarz abhängig ist [Sch17c].

Stromverbrauch

Der Bedarf an elektrischer Energie lässt sich, je nach Zweck, in verschiedene Modi klassifizieren:
Power mode untergliedert sich in active mode (Chip kann Pakete senden, empfangen oder aufzeichnen) mit einem Energieverbrauch zwischen 80 und 260mA in Abhängigkeit ob Wi-Fi oder Bluetooth aktiv ist und welche Aufgaben zugewiesen sind, dem modem-sleep mode (CPU und clock sind betriebsbereit und konfigurierbar, während Wi-Fi / Bluetooth deaktiviert sind) mit einem Verbrauch zwischen 3 und 20 mA (Nor-

[2] Besonders stromsparender Modus, lediglich RTC und ULP-Koprozessor wird betrieben. Wi-Fi und Bluetooth-Daten werden in der RTC gespeichert [Esp17d, S. 22]

mal: 5 ~10 mA), light-sleep mode (CPU pausiert bis dieser ein wake-up Signal erhält, während RTC und ULP-Koprozessor arbeiten) mit 0.8 mA, deep-sleep mode (lediglich RTC arbeitet, Wi-Fi- und Bluetooth-Verbindungs- daten werden in dieser Zeit in der RTC gespeichert, ULP-Koprozessor fungiert optional) mit einem Stromverbrauch zwischen 20 µA und 0.15 mA bei Betrieb des ULP-Koprozessors, sowie dem hibernation mode (Interner 8 MHz Oszillator, ULP- Koprozessor und RTC-Wiederherstellungsspeicher sind deaktiviert. Ein RTC timer und wenige RTC GPIOs[3] sind aktiv. Der RTC timer bzw. die RTC GPIOs können dem Chip ein wake-up Signal senden) mit 5 µA.
Indes teilt sich **Sleep Pattern** auf in association sleep pattern (power mode wechselt zwischen dem active mode und dem modem-sleep/light-sleep mode. CPU, Wi-Fi, Bluetooth bekommen in vorgegebenen Intervallen ein wake-up Signal um die WiFi- und Bluetooth-Verbindung aufrecht zu halten) und ULP sensor-monitored pattern (CPU befindet sich im deep-sleep mode, der ULP-Koprozessor führt Messungen an den Sensoren aus um dem System in Abhängigkeit der gemessenen Daten des Sensors ein wake-up Signal zu senden) [Esp17c, S. 8-9].
Mithilfe dieser Auflistung lässt sich ein geeigneter Modi für den entsprechenden Zweck zuordnen.

Peripheriegeräte und Sensoren

Der ESP32 unterstützt unter anderem zwei 12-bit Analog-Digital-Wandler, zwei 8-bit Digital-Analog-Wandler, zwei UARTs als serielle Schnittstellen, SD/SDIO/MMC Host Controller und weitere Interfaces [Esp17c, S. 9-12]. In diesem Abschnitt wird auf den SD/SDIO/MMC Host Controller näher eingegangen, da dieser zur Sicherung der Daten essenziell ist. Dieser Host Controller ermöglicht die Unterstützung von SD–Karten. So können Daten auf einem Flash-Speicher abgelegt und auch gelesen werden. Der Anschluss des ESP32 Mikrocontroller mit einer SD-Karte erfolgt dabei nach dem Muster der Tabelle 2.1.

Tabelle 2.1: Anschluss-Zuordnung von ESP32 mit SD-Karte [modifziert nach [Gra17]]

ESP32	**SD-Karten Pin**	**SPI Pin**
GPIO14 (MTMS)	CLK	SCK
GPIO15 (MTDO)	CMD	MOSI
GPIO2	D0	MISO
GPIO4	D1	N/C
GPIO12 (MTDI)	D2	N/C
GPIO13 (MTCK)	D3	CS
N/C	CD	N/C
N/C	WP	N/C

Nach einem Update vom Juli 2017 bietet das offizielle Espressif IoT Development Framework (kurz: ESP-IDF) neben SD/MMC Peripherie auch SPI Peripherie zum Verwenden

[3] Programmierbare Kontaktstifte, welche sowohl als Ein-, als auch als Ausgänge nutzbar sind.

einer SD-Karte an. Die Signale „card detect“ (CD) und „write protect“ (WP) werden in diesem Beispiel nicht verwendet. CLK (Clock) dient für die taktgesteuerte Datenübertragung. Über die Leitungen D0, D1, D2 und D3 (Data) werden die Daten transportiert. Die Transferrichtung erfolgt über CMD (Command) [Dem15, S. 75]. Durch setzen des Flags im Programmiercode mittels

```
host.flags = SDMMC_HOST_FLAG_1BIT;
```

ist es möglich, vom 4-bit Modus in den 1-bit Modus zu wechseln. Standardmäßig ist der 4-bit Modus gesetzt (zu finden in der Headerdatei sdmmc_host.h). Sobald D1, D2 oder D3 nicht genutzt werden, wird automatisch der 1-bit Modus verwendet. Der 4-bit Modus erlaubt es 4-bit-parallele Daten in den Flashspeicher der SD-Karte zu schreiben bzw. zu lesen.

Wi-Fi

Der Wi-Fi Chip unterstützt die Protokolle 802.11 b/g/n/e/i, die Protokoll-Daten- einheiten A-MPDU und A-MSDU, sowie 4 µs Guard Intervall (näheres zu Protokollen/Standards in den Abschnitten 2.3 und 2.3.3) im 2,4 GHz Frequenzband [Esp17c, S. 1]. Ein Guard Intervall dient der Verbesserung der Störfestigkeit bei der Datenübertragung. Sobald ein Signal den Empfänger erreicht, wird eine kurze Pause (das Guard Intervall) eingelegt, um nach Ablauf dieser Zeit das nächste Signal zu senden. Guard Intervall und Nutzdaten bilden zusammen ein Symbol. Je kürzer das Guard Intervall ist, desto höher ist der maximale Datendurchsatz [LAN17a].

2.2.4 Espressif IoT Development Framework

Der ESP32 wird mithilfe des Espressif IoT Development Frameworks (ESP-IDF) geflasht. Alternativ ist es auch möglich mit der Arduino IDE und der entsprechenden Board-Bibliothek den Chip zu flashen, jedoch bietet im Gegensatz dazu das ESP-IDF die Möglichkeit, diverse Einstellungen zu treffen. Diese Einstellungen werden über ein textbasiertes Konfigurationsmenü mittels dem Befehl

```
make menuconfig
```

getroffen und umfassen unter anderem:

- CPU Frequenz (80MHz, 160MHz, 240MHz)
- Aktivierung bzw. Deaktivierung von Watchdog und Brownout Detection, sowie Konfiguration dieser
- Panic handler (Bei Fehlermeldung diese ausgeben und Skript stoppen, diese ausgeben und ESP32 neustarten, ohne Meldung neustarten, ...)
- Wi-Fi Einstellungen (u.a. maximale Signalstärke in dBm)
- FAT Filesystem (Zeichensatztabelle, „Long filename support“ und maximale Zeichenlänge dieser)

- FreeRTOS[4] Einstellungen (FreeRTOS auf einem/beiden Kern(en) laufen lassen, Tickrate in Hz, ...)

Alle getroffenen Einstellungen werden in der Datei „sdkconfig“ im Projektordner gespeichert und bei jedem zukünftigen flashen des ESP32 verwendet. Die festgelegten Parameter können jederzeit im Entwicklerwerkzeug abgeändert werden. Ebenfalls kann die Datei „sdkconfig“ in jedem herkömmlichen Texteditor geöffnet und bearbeitet werden. Dabei ist zu beachten, dass der Dateiname unverändert bleiben muss. Sofern die angegebenen Parameter angepasst wurden, ist es notwendig den ESP32 erneut zu flashen, um die Änderungen wirksam auf dem Mikrocontroller zu hinterlegen.

Das ESP-IDF ist für Windows, Linux und Mac verfügbar und benötigt zum Kompilieren eine Toolchain. Die Toolchain ist eine „Kette von Werkzeugen“, welche benötigt werden, um den Quelltext in ein lauffähiges Programm zu überführen.

Für diese Arbeit wurde die von Espressif Systems empfohlene Xtensa Toolchain genutzt. Als Entwicklerwerkzeug diente MinGW/MSYS2[5]. Die aktuelle Version des ESP-IDFs kann über Github bezogen werden:

```
git clone --recursive https://github.com/espressif/esp-idf.git
```

Anschließend wird im Ordner `C:/msys32/etc/profile.d/` eine Datei `export_idf_path.sh` angelegt, welche den Pfad zum ESP-IDF beinhaltet.
Der Pfad wird in der Form
`export IDF_PATH="C:/msys32/home/user-name/esp/esp-idf"`
angegeben, sodass die Programme der Toolchain auf das ESP-IDF zugreifen können. Es ist ebenfalls möglich die Eclipse IDE anstatt `make` zu nutzen [Esp17b].

Innerhalb des Programmablaufes wird aufgrund der Syntax für Strings und GPIOs der „Arduino core“ genutzt. Dieser wird als Komponente innerhalb der Umgebung des ESP-IDFs über folgende Befehle im Projektordner eingebunden [Esp17a]:

```
mkdir -p components && \
cd components && \
git clone https://github.com/espressif/arduino-esp32.git arduino &&
    \
cd .. && \
make menuconfig
```

[4] Echtzeitbetriebssystem für eingebettete Systeme
[5] „All-in-one“-Lösung für Windows unter folgendem Link: https://esp-idf.readthedocs.io/en/latest/get-started/windows-setup.html, letzter Zugriff: 17.08.2017

Anschließend taucht unter `make menuconfig` folgender Punkt auf:

```
Component config --> Arduino Configuration --> Autostart Arduino
   setup and loop on boot
```

Dies hat weiterhin den Nebeneffekt, dass der Programmcode auch in der Arduino IDE kompiliert werden kann. Der Grund dafür ist unter anderem die teilweise erzwungene Verwendung von Funktionen wie `void setup()`, `void loop()` und die optionale Verwendung z.B. von `digitalWrite()` innerhalb des Programmcodes, welche auch von der Arduino IDE interpretiert werden können.

Neben `make menuconfig` umfassen die wichtigsten Befehle:

make all
Kompiliert den Quellcode, Bootloader und erstellt eine Partitionstabelle basierend auf den Einstellungen von `make menuconfig`.

make flash
Flasht das Projekt (Quellcode, Bootloader, Partitionstabelle) auf den ESP32 Chip. Einstellungen wie die serielle Schnittstelle und Baudrate[6] werden in `make menuconfig` getroffen.

make monitor
Ruft ein serielles Terminal auf, welches die Datenübertragung über die serielle Schnittstelle anzeigt.

make clean
Entfernt das Programm, den Bootloader und die Partitionstabelle vom ESP32 Chip. Dieser Befehl ist z.B. notwendig, wenn es während des Flash-Vorgangs zu Problemen kam bzw. dieser mittendrin abrupt abgebrochen wurde.

Auch eine Kombination der `make` -Befehle ist möglich, bspw. `make flash monitor` oder `make clean flash`

[6] Anzahl an Symbole pro Zeitspanne; 1 Baud entspricht dabei 1 Symbol pro Sekunde

2.3 IEEE 802.11-Standard

2.3.1 Hintergrund

IEEE 802.11 ist ein vom „**I**nstitutes of **E**lectrical and **E**lectronics **E**ngineers“ entwickelter Standard für drahtlose lokale Netzwerke. Im Jahr 1997 wurde die erste Version veröffentlicht. Mithilfe dieses Standards sollen Datenraten von 1 bis 2 Mbps im 2,4 GHz Frequenzbereich ermöglicht werden [Pre11, S. 1028]. Im Laufe der letzten Jahre wurde der Standard mehrmals weiter entwickelt, hierauf wird im Kapitel 2.3.3 genauer Bezug genommen. Der Datenrahmen (MAC Protocol Data Unit genannt) besteht aus dem MAC-Header, der MAC Service Data Unit (MSDU, auch Frame Body) und einem Prüfsummenfeld FCS [Sal16].
Des Weiteren spezifiziert der IEEE 802.11 Standard den Physical Layer und den MAC-Layer (Medium Access Control). Der Physical Layer stellt im OSI-Referenzmodell die Bitübertragungsschicht und MAC eine Unterschicht der Sicherungsschicht dar (siehe Anhang A.1). Die Bitübertragungsschicht ist für den Datentransport verantwortlich, während die MAC-Schicht der Koordinierung dient. Im 802.11 Standard verwendet die MAC-Schicht das CSMA/CA Protokoll (Carrier Sense Multiple Access without Collision Avoidance). Dieses Protokoll prüft, ob der Kanal für die Datenübertragung nicht belegt ist um eine Kollision mit anderen Datenpaketen zu vermeiden. Sender und Empfänger tauschen immer wieder Datenpakete aus, um die Empfangsbereitschaft zu überprüfen [Har13, S.9-10]. Die MAC-Schicht teilt sich die Sicherungsschicht mit der LLC-Schicht (Logical Link Control). Diese (Unter)Schicht ist als Schnittstelle für den Zugriff auf höhere Protokolle ausgelegt, wie beispielsweise Schicht 3: Vermittlungsschicht/Network Layer [Kau08, S. 206].

2.3.2 Aufbau des Frame Formats

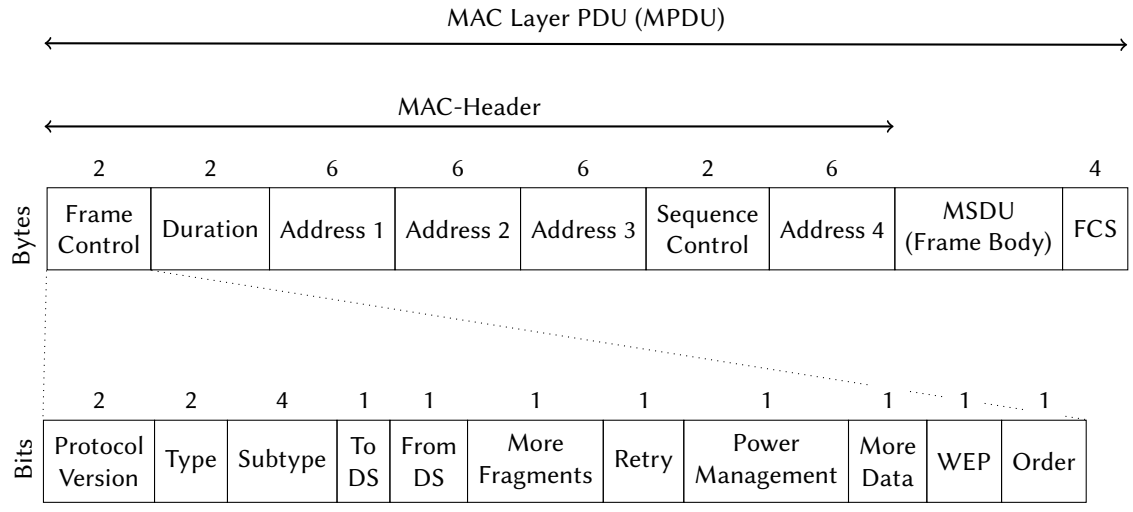


Abbildung 2.3: IEEE 802.11 Frame [modifiziert nach [Fin11, S. 77]]

Frame Control

Protocol Version
Beinhaltet die Version des 802.11-Protokolls, sodass der Empfänger eine potenzielle Unterstützung des Protokolls erkennen kann.

Type/ Subtype
Gibt die Aufgabe des Frames an: Management- (mgmt), Control- (ctrl), oder Data-Frame (data). Management-Frames teilen die Anforderungen und Attribute des Access Points mit. Control-Frames dienen der Steuerung des Mediums und dem Bestätigen von erhaltenen Paketen. Data-Frames beinhalten den Payload, also die eigentlichen Daten in Form einer MPDU [Bad15, S. 616-617].

To DS/ From DS
Dabei steht DS für „Distribution System“. Dies dient der Feststellung, ob das Paket das drahtlose Netzwerk verlässt, nicht verlässt oder diesem beitritt. Dies beeinflusst die Felder „Adresse 1“, „Adresse 2“, „Adresse 3“ und „Adresse 4“.

Tabelle 2.2: Erläuterung des Distribution Systems [modifziert nach [Fin11, S. 80]]

To DS	From DS	Beschreibung	Adresse			
			1	2	3	4
0	0	Frame direkt von einer Station an eine andere Station senden (Ad hoc- oder IBSS-Netzwerk).	DA	SA	BSSID	-
0	1	Frame von einem Access Point an einen Client senden (BSS- oder ESS-Netzwerk)	DA	BSSID	SA	-
1	0	Frame von einem Client an einen Acess Point senden (BSS- oder ESS-Netzwerk)	BSSID	SA	DA	-
1	1	Frame zwischen zwei Access Points in einer point-to-point bridge connection senden	RA	TA	DA	SA

In einem Ad-hoc Netzwerk (auch Peer-to-Peer-Netzwerk genannt) kommunizieren gleichberechtigt mindestens zwei Clients direkt (ohne Access Point) über ein Funknetz miteinander. Der 802.11 Standard bezeichnet dieses Netzwerk als Independent Basic Service Set (IBSS). BSS (Basic Service Set) stellen Netzwerke mit nur einem Access Point und einem oder mehreren Clients dar. Sobald ein BSS mehr als einen Access Point enthält, bezeichnet man dies als ESS (Extended Service Set) [Fra08, S. 12-13]. Die Destination Address (DA) ist das Ziel des Frames, wo dieser verarbeitet wird. Der Frame wird von der Source Address (SA) erstellt. Die Basic Service Set ID (BSSID) stellt die MAC-Adresse des Access Points dar.

Da es in Ad-hoc Netzwerken keine Access Points gibt, wird diese 48-bit lange Adresse zufällig generiert [Fin11, S. 12].
Transmitter (TA) sind Stationen, welche den Frame an das drahtlose lokale Netzwerk senden, jedoch nicht der ursprüngliche Ersteller des Frames sind. Die Station, welche den Frame über das Funknetzwerk empfängt, jedoch nicht die Destination Address ist, wird als Receiver Address (RA) bezeichnet [Fin11, S. 80].

More Fragments
Zeigt an, ob das Paket in mehrere Fragmente aufgeteilt ist. Dies geschieht, wenn der Frame zu groß für ein Paket ist.

Retry
Sofern der Bit auf 1 gesetzt ist, wurde der Frame erneut gesendet.

Power Management
Gibt an, ob der Sender im Stromsparmodus agiert, sodass der Energieverbrauch reduziert wird.

More Data
Weist einen Client im Stromsparmodus darauf hin, diesen nicht zu verlassen, da der Access Point weitere gepufferte Daten besitzt.

WEP
Sobald der Bit auf 1 gesetzt wird, sind die Information im Datenfeld Frame Body verschlüsselt.

Order
Kennzeichnung, dass die Frames in einer bestimmten Reihenfolge gesendet werden.

Duration
Dieses Feld besitzt zwei Funktionen: Die Zeit (in Millisekunden) für die Übertragung des Frames. Dies ist nowendig, da der Frame Body eine variable Länge besitzt. Da in dieser Zeit das Medium beschäftigt ist, hören die Access Points auf, auf Pakete zu warten. Des Weiteren senden Access Points nach Ablauf dieser Zeit Clients, welche aus dem Stromsparmodus erwachen, die gepufferten Daten zu.

Adress 1, 2, 3, 4
Je nach Kombination aus Typ und Subtyp des Frames nehmen diese Adressen unterschiedliche Aufgaben an (siehe Tabelle 2.2). Normalerweise werden 3 MAC-Adressen (Source Adress, Destination Adress, BSSID) genutzt, Adresse 4 ist optional und wird nur bei Data-Frames eingesetzt [Hof05].

Sequence Control

Sequence Control untergliedert sich in Fragment Number (4 Bit) und Sequence Number (12 Bit). Pakete, welche aufgrund Ihrer Größe in Fragmente zerlegt werden, erhalten eine eindeutig zuweisbare Fragmentnummer. Diese beginnt bei 0 und erhöht sich für jedes Fragment um den Wert 1. Dies dient dem späteren zusammensetzen der Fragmente. Die Sequenznummer wird jedem Datenpaket eindeutig zugewiesen um Duplikate zu erkennen. Für jedes Datenpaket wird die Sequenznummer ebenfalls um 1 erhöht [Bal12].

MSDU (Frame Body)

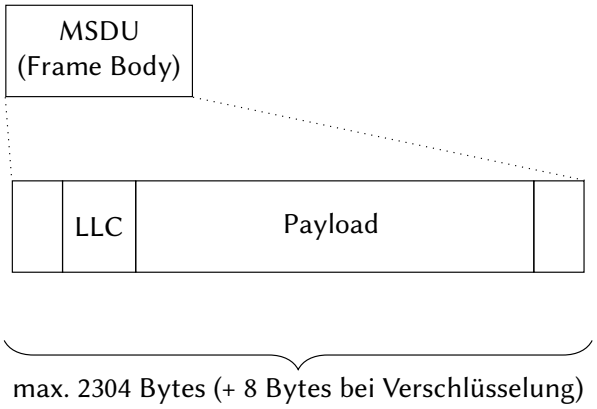


Abbildung 2.4: Frame Body [modifiziert nach [Bad15, S. 617]]

Die MSDU (MAC Service Data Unit) setzt sich zusammen aus Logical Link Control und dem Payload (Datenanteil). Je nach Art der genutzten Verschlüsselung (WEP, WPA, WPA2) ergänzt sich die Struktur.
Weiterhin wird zwischen A-MPDU und A-MSDU differenziert, wobei A für aggregated steht. Ein A-MPDU sind angehäufte MPDUs welche an den PHY-Layer weitergegeben und übertragen werden. Die maximale Größe eines A-MPDU beträgt 65.535 Bytes. Gleiches gilt für A-MSDU, wobei mehrere MSDUs zu einem A-MSDU zusammengefasst werden. Die maximale Größe ist dabei limitiert auf 7.935 Bytes. Mit diesem Verfahren können mehrere Frames gleichzeitig versendet werden [LC08, S. 557-558].

WEP: Ein mittlerweile als unsicher geltendes Verschlüsselungsprotokoll, welches RC4 Stromchiffren-Verschlüsselung mit einem 40-bit bzw. 104-bit langen Schlüssel benutzt. Die linke Seite der MSDU wird ergänzt durch 24-bit Initialisierungsvektor (dieser wird zur Verschlüsselung mit dem 40-bit bzw. 104-bit Schlüssel kombiniert), 6-bit Padding und 2-bit Key-ID (dient dem Empfänger zusammen mit dem Initialisierungsvektor zur Entschlüsselung). Die rechte Seite ergänzt sich durch 32-bit ICV (Integrity Check Value) und dient als Prüfsumme.

WPA: Nutzt das Temporal Key Integrity Protocol zur Absicherung, dabei wird jedes Paket mit einem anderen Schlüssel abgesichert. Wie WEP nutzt auch WPA RC4 Stromchiffren. Zusätzlich wird Michael-MIC (fortlaufende Nummerierung der Pakete) für eine verbesserte Integritätsprüfung verwendet. Die linke Seite der MSDU besteht hierbei aus einem 32-bit Initialisierungsvektor (IV) und einer Key-ID, sowie 32-bit Extended IV. Die rechte Seite wird durch 64-bit Michael-MIC und 32-bit ICV ergänzt.

WPA2: Dieses Verfahren nutzt, basierend auf blockweise AES-Verschlüsselung, CCMP (Counter Mode with Cipher Block Chaining Message Authentication Code Protocol) für den Payload. Ergänzt wird die linke Seite durch 64-bit CCMP Header, die rechte Seite durch 64-bit MIC. WPA2 gilt aktuell als das sicherste Verschlüsselungsverfahren im Wireless-Bereich [Bad15, S. 619].

FCS

FCS steht für „Frame Check Sequence“ (Blockprüfzeichenfolge). Die Sendestation berechnet aus dem MAC-Header, sowie dem Frame Body einen binären Wert und legt diesen im Datenfeld FCS ab. Nach dem Erhalt berechnet der Empfänger mit dem selben Algorithmus ebenfalls eine Prüfzeichenfolge und gleicht diesen mit dem hinterlegten Wert ab um sicherzustellen, dass es zu keinen Übertragungsfehlern kam [Sal16].

2.3.3 Kanäle und Frequenzen

Die Norm IEEE 802.11 unterscheidet sich im Frequenzbereich und den dazugehörigen Kanälen in den verschiedenen Standards. Gängige Frequenzbänder sind 2,4000 - 2,4835 GHz, sowie 5,150 - 5,725 GHz (abhängig vom Land, da es Unterschiede zwischen Europa, USA und Japan gibt) [Bau15, S. 48]. Jeder Standard nutzt ein Modulationsverfahren für die Kanalbreite und den Kanalrasterabstand. Je nach Modulationsverfahren, Frequenzblock und Standard kann es somit zur Überlappung von Signalen kommen. Beispielsweise nutzt der 802.11b Standard den Frequenzblock 2,4 GHz, das Modulationsverfahren DSSS[7] und eine Kanalbreite von 22 MHz. Der Kanalrasterabstand beträgt hierbei 5 MHz, somit überlappen sich die Signale bis auf die der breitbandigen Kanäle 1, 6 und 11 (denkbar ist auch die überlappungsfreie Nutzung der Kanäle 1, 7 und 13) [Bau15, S. 50]. Dies wird in Abbildung 2.5 veranschaulicht. Da im 5 GHz Frequenzband sowohl eine Kanalbreite von 20 MHz, 40 MHz, 80 MHz als auch 160 MHz (je nach verwendeten Standard) genutzt werden kann, sind die überlappungsfreien Kanäle abhängig von der genutzten Kanalbreite bzw. des Standards [Lan17b, S. 365].

Welches Frequenzband und welches Modulationsverfahren verwendet werden, ist vom genutzten Standard abhängig. So nutzen die IEEE-Standards 802.11, 802.11b und 802.11g 2,4 GHz, während 802.11a und 802.11h 5 GHz nutzen.

[7] Frequenzspreizverfahren, welches Daten über einen breiten Frequenzbereich verteilt und daher unempfindlich gegenüber schmalbandigen Störungen (u.a. Bluetooth) ist

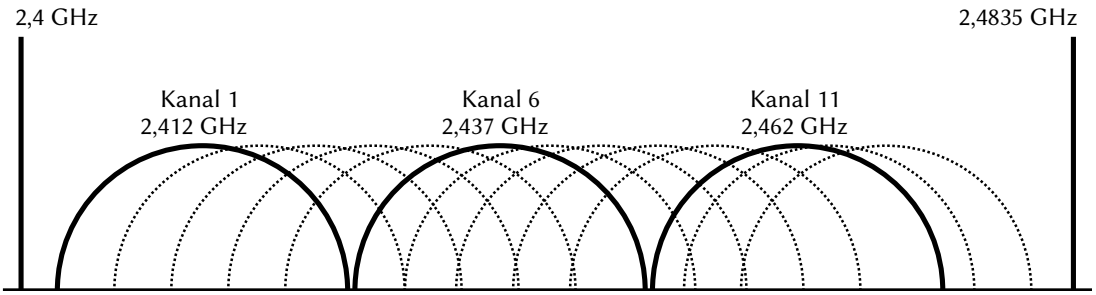


Abbildung 2.5: Kanalverteilung des 802.11b Standards [modifiziert nach [Bau15, S. 51]]

Der Standard 802.11n kann dagegen sowohl 2,4 GHz, als auch 5 GHz verwenden.
Die verwendbaren Kanäle sind abhängig vom Land, in dem der Standard genutzt wird. In Europa sind insgesamt 13 Kanäle im 2,4 GHz Frequenzband nutzbar (2,4000 GHz - 2,4835 GHz; á 22 MHz Kanalbreite bei den Standards 802.11 und 802.11b bzw. á 20 MHz Kanalbreite bei den Standards 802.11a, 802.11g, 802.11h und 802.11n, letzteres optional auch mit einer Kanalbreite von 40 MHz bei Verwendung im 5 GHz Frequenzband), während in den USA 11 Kanäle zur Verfügung stehen (Einschränkung des Frequenzbands von 2,4000 GHz - 2,4745 GHz) und in Japan ein zusätzlicher 14. Kanal existiert (12 MHz über den 13. Kanal im 2,4 GHz Frequenzband) [Bau15, S. 48 - 50].
Ziel der verschiedenen Modulationsverfahren und Kanäle ist es, eine möglichst störungsfreie Verbindung aufzubauen. Beispielsweise nutzen Mikrowellenherde zumeist die Frequenz 2,455 GHz, was den Kanälen 9 und 10 im 2,4 GHz Frequenzband entspricht [Kam12, S. 123]. Auch Bluetooth sendet in diesem Frequenzbereich und nutzt den Frequenzblock 2,402 - 2,480 GHz, jedoch wird hier dank des genutzten Modulationsverfahren bis zu 1600 mal pro Sekunde der Kanal gewechselt, um Störungen zu vermeiden [Bau15, S. 55].

2.3.4 Sniffing in drahtlosen Netzwerken

802.11-Adapter und deren Treiber filtern Pakete auf der Empfängerseite. Eine Variante ist eine Filterung über die im Kapitel 2.3.3 erwähnten Kanäle. Bei solchen Adaptern bzw. auch bei Access Points wählt der Nutzer die entsprechenden Kanäle aus, auf denen der Datenverkehr empfangen oder gesendet werden soll. Es kann nur solcher Datenverkehr verarbeitet werden, welcher über diesen Kanal gesendet oder empfangen wurde.
Eine weitere Variante ist das Filtern über den Netzwerknamen (SSID / ESSID, siehe Erläuterung zur Tabelle 2.2). Die SSID wird vom Nutzer am Access Point und dem Netzwerkadapter ausgewählt. Sobald mehrere Access Points die gleiche SSID nutzen, spricht man von einem extended SSID (ESSID).
Des Weiteren wird nach der MAC-Adresse gefiltert. Dabei filtert der Netzwerkadapter alle empfangenen Pakete nach der Ziel-MAC-Adresse (siehe Tabelle 2.2, Destination Address) und gibt an den Host folgendes weiter [Wir17d]:

- alle Unicast-Pakete, welche *an eine Adresse* des Adapters gesendet wurden (beispielsweise Pakete welche, an den Host des Netzwerks gesendet wurden). Unicast-Pakete sind alle Pakete, welches an ein Ziel gesendet werden [Wir17c].
- alle Multicast-Pakete, welche an eine Multicast-Adresse des Adapter gesendet werden oder aber auch alle Multicast-Pakete, egal an welche Adresse diese gesendet werden. Dies ist abhängig von dem genutzten Netzwerk-Adapter und deren Konfiguration [Wir17d]. Multicast erlaubt das Senden eines einzelnen Pakets *an mehrere Empfänger* [Wir17b].
- alle Broadcast-Pakete. Diese Pakete werden von einem Sender *an alle Teilnehmer* des Netzwerks gesendet [Wir17a].

2.3.4.1 Promiscuous Mode

Sofern die Netzwerkkarte in den Promiscuous Mode (auch promiskuitiver Modus genannt) versetzt wurde, ist es möglich, allen Daten mitzulesen, welche an den Netzwerkadapter gesendet wurden. Dies bedeutet, dass eine aktive Verbindung mit dem Netzwerk hergestellt werden muss, da Datenpakete aus fremden Netzwerken ignoriert werden. Alle Pakete werden an den Client weitergeleitet, dessen Netzwerkkarte in diesen Modus versetzt wurde [MS11, S. 172].
Der im Abschnitt 2.3.4 erwähnte MAC-Adressen Filter wird deaktiviert, sodass alle Pakete des aktuellen Netzwerks aufgezeichnet werden können. Wie im Abschnitt 2.3 erwähnt, entsprechen MAC-Adressen im OSI-Modell der Sicherungsschicht (Data Link Layer). Hierfür muss die SSID und der Kanal korrekt hinterlegt sein. Weiterhin ist zu beachten, dass Pakete, welche ihren Ursprung aus einem geschützten Netzwerk haben, nicht vom Netzwerk-Adapter entschlüsselt und aufgezeichnet werden können [Wir17d].

2.3.4.2 Monitor Mode

Der Monitor Mode dient zum Empfangen von Paketen aus fremden Netzwerken, ohne dass eine aktive Verbindung mit dem Netzwerk besteht [MS11, S. 172]. In diesem Modus wird der im Abschnitt 2.3.4 angeführte SSID Filter deaktiviert, sodass alle Pakete von allen SSIDs des benutzten Kanals aufgezeichnet werden. Dies entspricht der Bitübertragungsschicht (Physical Layer) im OSI-Modell. Abhängig von Netzwerk-Adapter und Treiber kann es zu einem Verbindungsabbruch mit dem aktuellen Netzwerk kommen. Sofern es zu einem solchen Abbruch kommt und der Host keine weiteren Netzwerk-Adapter besitzt, um sich mit einem Netzwerk zu verbinden, ist es auch nicht mehr möglich, Adressen in einen Hostnamen aufzulösen, Pakete als Dateien auf einem Network File System (ermöglicht Zugriff auf Daten über ein Netzwerk) abzulegen und alles weitere, was eine aktive Verbindung erfordert [Wir17d]. [8]

[8] Anmerkung: In der Dokumentation, sowie des Quellcodes des ESP-IDFs wird der Begriff „Promiscuous Mode“ verwendet, dies wird in dieser Arbei beibehalten. Es wird hier darauf hingewiesen, dass es sich dabei um den „Monitor Mode“ handelt, welcher in diesem Projekt auch verwendet wird.

3 Materialien und Methoden

3.1 Schaltplan

Der Schaltplan (Abbildung 3.3) wurde mithilfe der Software „KiCad“ erstellt.

Den Mittelpunkt des Schaltplans stellt der ESP32 (in diesem Fall das Development Board) dar. An diesem befindet sich am 3.3V GPIO eine Leitung, welche zum SD-Karten Adapter führt. An dieser befindet sich mit Pullup-Widerständen die Kontaktpins für DAT2, CD/-DAT3, CMD, CLK, DAT0 und DAT1 (Abschnitt 2.1). Bei einem Pullup-Widerstand wird dieser mit dem Eingangspin (in diesem Fall der SD-Karte) und der Versorgungsspannung (3.3V) verbunden. Damit sind die entsprechenden Pins der SD-Karte dauerhaft auf „HIGH“ (logisch „1“) gesetzt.

Der ESP32 besitzt ebenfalls, wie viele derartiger Module, interne Pull-Up Widerstände, welche über die Software hinzugeschalten werden können:
`digitalWrite(GPIO, HIGH)` (bzw. LOW) innerhalb der Arduino IDE und
`gpio_set_level(GPIO, 1)` (bzw. 0), innerhalb des ESP-IDFs unter der Programmiersprache C und `gpio_set_level((gpio_num_t)GPIO, 1)` (bzw. 0) unter der Programmiersprache C++. Eine Ausnahme stellt das Einbinden des „Arduino cores“ im ESP-IDF dar, dazu mehr im Abschnitt 2.2.4.
Die Spannungsversorgung des VDD-Pins der SD-Karte erfolgt ebenfalls über den 3.3V Pin. Beim Schreibvorgang der SD-Karte werden kurzzeitige Spitzen in der Stromaufnahme verursacht. Damit der anschließende Spannungsabfall über die Zuleitung nicht zu groß wird, wird ein Kondensator (hier 100nF Keramikkondensator) so dicht wie möglich am Verbraucher angebracht. Der Kondensator sollte klein gewählt werden, da größere Kondensatoren langsamer Energie abgeben.
An den GPIOs 5, 18 und 19 befindet sich jeweils eine LED mit einem 100Ω Widerstand. Diese LEDs visualisieren verschiedene Meldungen: LED_PAKET blinkt für jedes empfangene Paket auf. LED_READY blinkt mehrmals kurz auf, sobald das Programm per Drucktaster gestartet wurde und die Initialisierung des Programmcodes erfolgreich war. LED_ERROR signalisiert Fehlermeldungen, welche die SD-Karte betreffen (Fehler beim Mounten oder Schreiben). Hierzu mehr im Abschnitt 3.2.1.

GPIO21 befindet sich über einen 1kΩ Widerstand an der Basis eines NPN Transistors (BC547B) und dessen Collector befindet sich am 5V Pin des ESP32. Sobald GPIO21 auf HIGH (logisch „1“) gesetzt wird, schaltet der Transistor durch und schließt den Stromkreis. Dies erfolgt vor der Initialisierung des Programmcodes innerhalb eines vordefinierten Zeitraums für 50ms und dient einer Powerbank (Spannungsversorgung) als Wake-Up Signal. Dies wird im Abschnitt 3.3 näher erläutert.

Weiterhin ist ein Drucktaster mit GPIO23, dem 5V Pin und über einen 10kΩ Widerstand mit GND des ESP32 verbunden. Der Drucktaster startet bei einmaligen Drücken das Programm und beendet dieses bei einem weiteren Druck. Der aktuelle Zustand des Drucktasters wird im Programmablauf abgerufen. Über eine weitere Leitung werden die drei LEDs, der Transistor, der Drucktaster und die SD-Karte an GND des ESP32 geführt. Ein möglichst platzsparender Aufbau ist den Abbildungen 3.1 und 3.2 zu entnehmen.

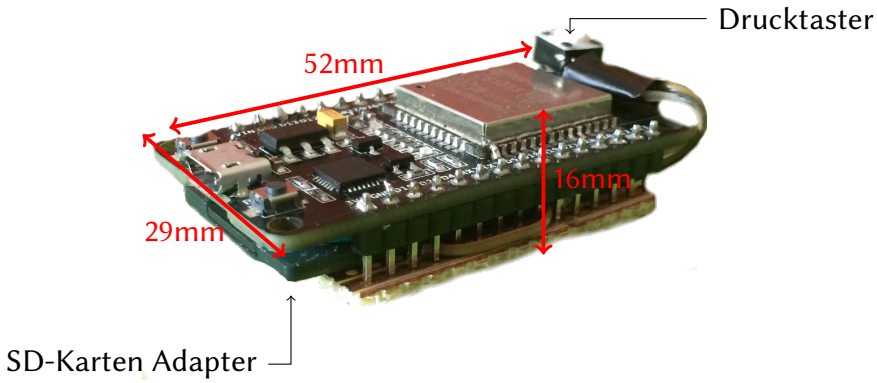


Abbildung 3.1: Platzsparender Aufbau des embedded Device

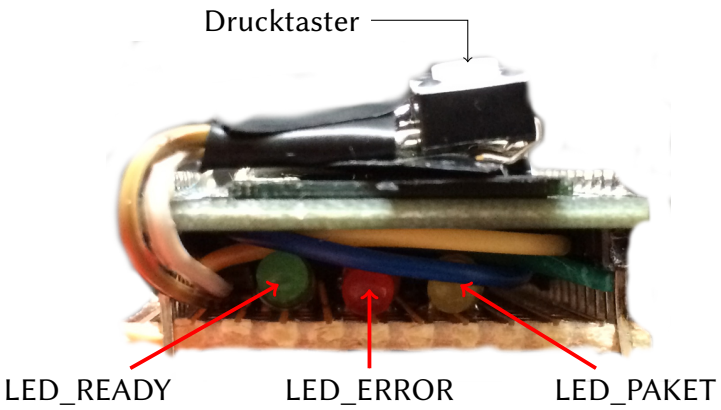


Abbildung 3.2: Seitenansicht des embedded Device

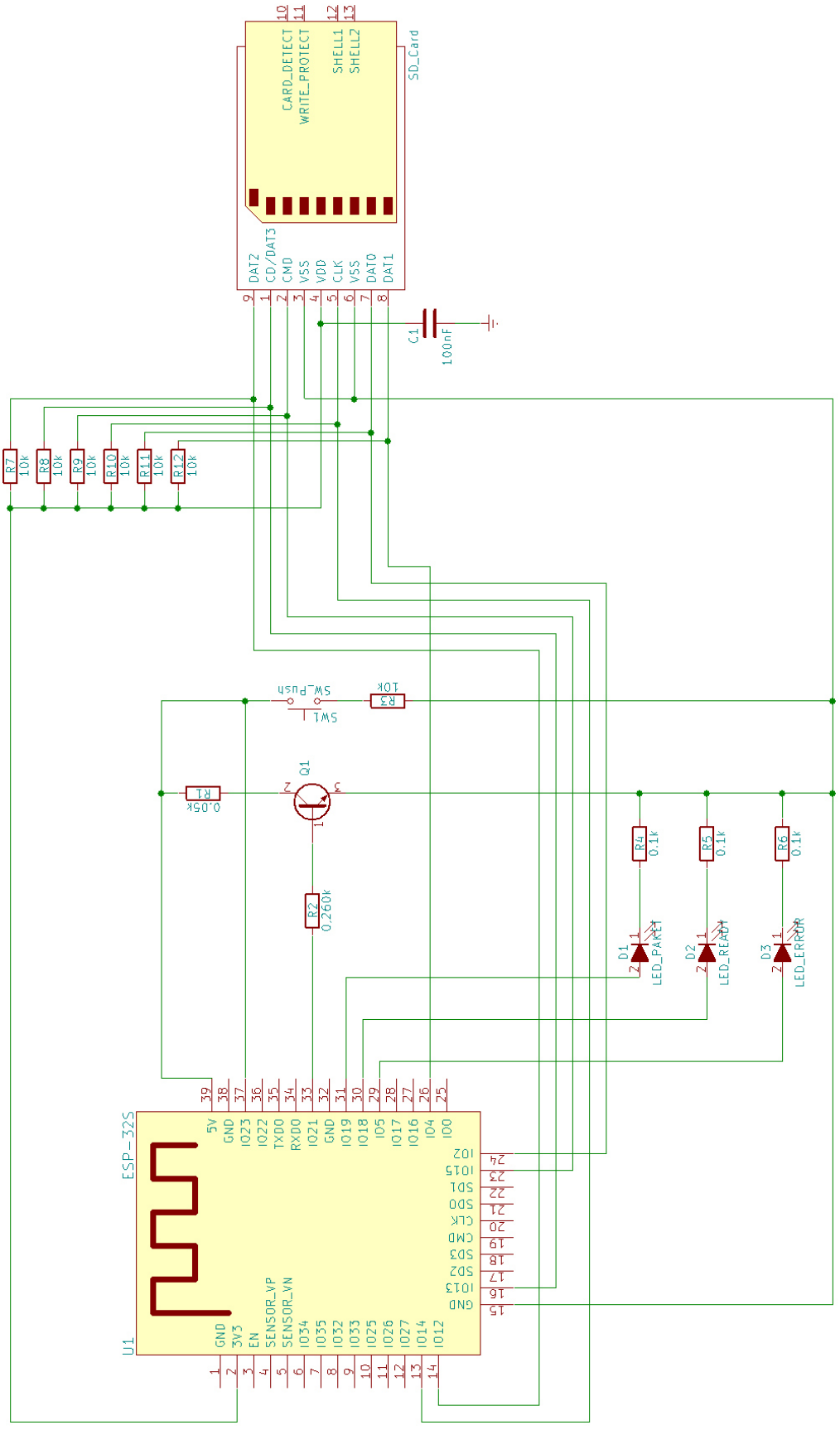


Abbildung 3.3: Schaltplan: Embedded Device (ESP32) zur Aufzeichnung von Drahtlosnetzwerkverkehr

3.2 Programmablauf

3.2.1 Parameterübergabe

Essenzielle Parameter, welche für die korrekte Verwendung des Programms notwendig sind, werden in den Zeilen 32 bis 44 deklariert:

```
32 //////////////////////////EINSTELLUNGEN/////////////////////////
33 #define FILENAME "benchmark" // Dateiname festlegen (ohne
       Dateiendung)
34 #define SWITCHING_TIME 50 // Zeit in ms, bis der Kanal gewechselt
       wird
35 #define CHANNEL_SWITCHING true // True, wenn Kanal automatisch
       gewechselt werden soll
36 #define CHANNEL 1 // Kanal welcher gesnifft werden soll (wenn
       CHANNEL_SWITCHING false)
37 #define LED_ERROR 5 // LED-Pin fuer Fehlermeldung definieren
38 #define LED_READY 30 // LED-Pin fuer erfolgreiche Initialisierung
       definieren
39 #define LED_PAKET 19 // LED-Pin fuer empfangene Pakete definieren
40 #define BUTTON 23 // Drucktaster-Pin fuer Start/Stop des Skripts
41 #define POWERBANK 21 // Pin fuer die Basis des Transistors
42 #define DEAUTHENTIFICATION false // True, wenn eapol 4-way
       handshake benoetigt wird (Deauth-Attacke)
43 uint8_t mac_filter[] = { }; // sniff specific mac adress
44 // Fuer Adresse 1-4, Form (Hex): { 0xff, 0xff, 0xff, 0xff, 0xff,
       0xff, }. Filter-Deaktivierung mit: { }
45 ////////////////////////////////////////////////////////////////
```

Programmcode 3.1: Festlegung der Parameter; esp32_promiscuous.cpp (C++)

FILENAME
Hier erfolgt die Festlegung des Dateinamens der Capture-Datei als String. Diese Variable wird ohne Dateiendung angegeben, da diese (.pcap) im Zuge des Programmablaufes (Zeile 81) selbstständig ergänzt wird.

SWITCHING_TIME
Die Interpretation des Parameters erfolgt als Millisekunden. Nach Ablauf dieses Intervalls wird der Kanal gewechselt, siehe Unterpunkt CHANNEL_SWITCHING.

CHANNEL_SWITCHING
Sollen alle Kanäle im 2,4 GHz Frequenzband beachtet werden, wird diese Variable auf „true“ gesetzt. In diesem Fall wird automatisch in einem definierten Zeitintervall der Kanal von 1 an inkrementiert, bis der Wert 13 ist und anschließend wieder auf 1 zurückgesetzt. Dies erfolgt in einer while(true)-Schleife, sodass der ESP32 autark alle Kanäle von 1 bis 13 automatisch durchwechselt. Der Zeitintervall wird im Unterpunkt „CHANNEL_TIME“ definiert.

CHANNEL

Falls nicht alle 13 Kanäle gesnifft werden sollen, sondern lediglich ein bestimmter, kann dieser hier als Wert übergeben werden. Es gilt zu beachten, dass in diesem Fall der Parameter CHANNEL_SWITCHING auf false gesetzt werden muss. Andernfalls wird der übergebene Wert ignoriert.

LED_ERROR

Übergabe des Kontaktpins, an welchem sich eine LED zur visualisierten Fehlermeldung befindet. Die Fehlermeldungen umfassen:

- Fehler beim mounten der SD-Karte

```
249         ESP_LOGE(TAG, "Mounten des Dateisystems
                fehlgeschlagen. "
250                 "Wenn die SD-Karte automatisch formatiert
                        werden soll -> format_if_mount_failed =
                        true.");
251         digitalWrite(led1, HIGH);
252         vTaskDelay(500 / portTICK_PERIOD_MS);
253         digitalWrite(led1, LOW);
254         ESP.restart();
255       }
256       else {
257         ESP_LOGE(TAG, "Initialisieren der SD-Karte
                fehlgeschlagen (%d). ", ret);
258         digitalWrite(led1, HIGH);
259         vTaskDelay(500 / portTICK_PERIOD_MS);
260         digitalWrite(led1, LOW);
261         ESP.restart();
262       }
263     }
264     //SD-Karte erfolgreich initialisiert. Name, Typ,
            Speicherplatz, etc im Serial Monitor ausgeben
265     sdmmc_card_print_info(stdout, card);
```

Programmcode 3.2: Mounten der SD-Karte fehlgeschlagen; esp32_promiscuous.cpp (C++)

- Fehler beim Öffnen und damit anschließendem Schreiben der Datei

```
94    sprintf(filenamebuf, "/sdcard/%s.pcap", filename);
95    //ESP_LOGI(TAG, "Opening file");
96    FILE* f = fopen(filenamebuf, "a");
97
98    if (f == NULL) {
99      ESP_LOGE(TAG, "Fehler! Datei konnte nicht geoeffnet
            werden!");
```

Programmcode 3.3: Öffnen der SD-Karte fehlgeschlagen; esp32_promiscuous.cpp (C++)

LED_READY

Übergabe des Kontaktpins, an welchem sich eine LED zur Visualisierung der erfolgreichen Initialisierung befindet. Sobald das Programm ausgeführt wird und alle notwendigen Parameter erfolgreich initialisiert sind, blinkt diese LED mehrmals kurz auf. Beim Beenden des Programms leuchtet die LED einmal kurz auf.

LED_PAKET

Übergabe des Kontaktpins, an welchem sich eine LED zur Visualisierung jedes empfangenen Pakets befindet. Auch wird diese LED genutzt um das Wake-up Signal der Powerbank zu veranschaulichen. Weiteres zu diesem Thema folgt im Abschnitt 3.3.

mac_filter

Sollte ein spezifischer Access Point gesnifft werden, muss dieser bei MAC-Filter hinterlegt werden. Hier ist die korrekte Einhaltung der Syntax notwendig. Die MAC-Adresse wird dabei in folgendem Format übergeben:

```
{ 0xAA, 0xAA, 0xAA, 0xAA, 0xAA, 0xAA }
```

Dies entspricht der MAC-Adresse AA:AA:AA:AA:AA:AA. Die Syntax ist auf den Promiscuous Mode des ESP32 zurückzuführen. Alle empfangenen Pakete sind hexadezimal codiert. Um einen korrekten Abgleich der empfangenen Pakete mit dem MAC-Filter zu gewährleisten, wird der MAC-Filter in der identischen Syntax übergeben. Falls im MAC-Filter keine MAC-Adresse hinterlegt sein sollte, werden alle Access Points im Empfangsbereich gesnifft. In diesem Fall wird der Parameter mit

```
{ }
```

übergeben. Sollte die MAC-Adresse des gewünschten Access Points unbekannt, jedoch die SSID bekannt sein, ist es möglich zuerst den MAC-Filter frei zu lassen und einige Sekunden zu sniffen. Anschließend kann der
Hexdump in Wireshark analysiert werden. Es sollten vom Access Point u.a. Beacon Frames aufgezeichnet worden sein. In diesen befindet sich neben der MAC-Adresse auch die aufgelöste SSID um so dieses Paar zu verknüpfen. Die nun vorhandene MAC-Adresse wird nun im MAC-Filter übergeben und der ESP32 erneut geflasht. Anschließend werden alle Pakete, welche sich im Empfangsbereich befinden, nach dieser MAC-Adresse gefiltert und aufbereitet.

3.2.2 Initialisierung

Die Initialisierung wird in der Funktion `void Initialisierung()` vorgenommen. Dies bedeutet im Detail:

```
231   sdmmc_host_t host = SDMMC_HOST_DEFAULT();
232   host.max_freq_khz = SDMMC_FREQ_HIGHSPEED;
233
234   // To use 1-line SD mode, uncomment the following line:
235   host.flags = SDMMC_HOST_FLAG_1BIT;
236
237   //Standard-Initialisierung beachtet nicht CD und WP Pins bei
          SD-Karte
238   //Sollten diese genutzt werden muss slot_config.gpio_cd und
          slot_config.gpio_wp modifiziert werden
239   sdmmc_slot_config_t slot_config = SDMMC_SLOT_CONFIG_DEFAULT();
```

Programmcode 3.4: Initialisieren der SD-Karte; esp32_promiscuous.cpp (C++)

Die Funktion `#define SDMMC_HOST_DEFAULT` aus der Headerdatei `sdmmc_host.h` wird mit allen getroffenen Einstellungen wie GPIO Spannung, max. Frequenz in KHz, 4-bit/ 1-bit Mode (Abschnitt 2.2.3 unter Peripheriegeräte und Sensoren), usw., übernommen.
Die Variable `host.max_freq_khz` wird mittels `SDMMC_FREQ_HIGHSPEED` angepasst. Dies ermöglicht die Erhöhung der Taktrate von 20.000 KHz auf 40.000 KHz [Esp17f]. Dieses Feature ist jedoch in der aktuellen Version noch nicht fest implementiert, deshalb wird mit der Standard (default) Geschwindigkeit gelesen, sowie geschrieben. Im Abschnitt 4.2.2 wird dies veranschaulicht.
Des Weiteren wird `host.flags` in den 1-bit Modus versetzt. Durch Auskommentieren der Zeile 235 wird der 4-bit Modus verwendet.

```
267   tcpip_adapter_init();
268   wifi_init_config_t cfg = WIFI_INIT_CONFIG_DEFAULT();
269   ESP_ERROR_CHECK(esp_wifi_init(&cfg));
270   ESP_ERROR_CHECK(esp_event_loop_init(event_handler, NULL));
271   ESP_ERROR_CHECK(esp_wifi_set_storage(WIFI_STORAGE_RAM));
272   ESP_ERROR_CHECK(esp_wifi_set_mode(WIFI_MODE_NULL));
273   ESP_ERROR_CHECK(esp_wifi_start());
274
275   esp_wifi_set_channel(channel, WIFI_SECOND_CHAN_NONE);
```

Programmcode 3.5: Wi-Fi Initialisierung; esp32_promiscuous.cpp (C++)

`tcpip_adapter_init();` initialisiert den TCP/IP Adapter, welcher als Schnittstelle zwischen TCP/IP Stack, event handler, Wi-Fi Treiber und Ethernet Treiber dient. Die folgenden Zeilen überprüfen die Initialisierung des Wi-Fi Treibers und geben bei einem Fehler eine entsprechende Meldung aus.
`esp_wifi_set_channel(channel, WIFI_SECOND_CHAN_NONE);` legt den Kanal fest, auf welchem gesnifft werden soll.

```
285     xTaskCreatePinnedToCore(
286     preparing_hexdump, /* Function to implement the task */
287     "preparing_hexdump", /* Name of the task */
288     8192, /* Stack size in words */
289     NULL, /* Task input parameter */
290     1, /* Priority of the task */
291     NULL, /* Task handle. */
292     1); /* Core where the task should run */
```

Programmcode 3.6: Task auf zweiten Kern erstellen; esp32_promiscuous.cpp (C++)

`xTaskCreatePinnedToCore` erstellt einen `Task` auf dem zweiten Kern. Der zweite Kern wird mit 1 deklariert, der erste Kern mit 0.
Dieser Task beinhaltet die Funktion `void preparing_hexdump` in Zeile 111. Über diesen Weg werden die Aufgaben im Programmablauf auf die beiden Kerne aufgeteilt: Kern 1 übernimmt die Initialisierung, Wi-Fi (inkl. Promiscuous Mode), die Callback-Funktion und das Speichern auf SD-Karte (+ Deauthentification Attack), während Kern 2 den Callback in einen Hexdump aufbereitet und an die Funktion `void save_to_sdcard` übergibt. Andernfalls würde während des Aufbereitens des Hexdumps der Buffer überlaufen und Pakete verworfen werden, daher wird hier außerdem ein Ringpuffer verwendet, welcher im Abschnitt 3.2.3 erläutert wird.

```
294   esp_wifi_set_promiscuous_rx_cb(&wifi_promiscuous);
295
296   if (DEAUTHENTIFICATION) deauthentification_attack();
297
298   esp_wifi_set_promiscuous(true);
299   Init_success = 1;
```

Programmcode 3.7: Aktivieren des Promiscuous Modes; esp32_promiscuous.cpp (C++)

Sofern in Zeile 41 `DEAUTHENTIFICATION` den Wert `true` annimmt, wird die Funktion `void deauthentification_attack()` aufgerufen (siehe Abschnitt 3.2.4).
`esp_wifi_set_promiscuous_rx_cb(&wifi_promiscuous)` aktiviert den Callback in der Funktion `void wifi_promiscuous`, welcher bei jedem empfangenen Paket aufgerufen wird.
Der eigentliche Promiscuous Mode wird durch `esp_wifi_set_promiscuous(true)` aktiviert. Sofern die Variable `Init_success1` den Wert „1“ annimmt, wird dem restlichen Programmablauf die abgeschlossene Initialisierung signalisiert.

Die Funktion `void loop()` übernimmt drei verschiedene Aufgaben:

```
307    //Abfrage ob Drucktaster betaetigt wurde -> Wenn ja, rufe
           Funktion "Initialisierung" auf
308    buttonState = digitalRead(button);
309    if (buttonState == HIGH) {
310       if (Init_success == 0) {
311          Initialisierung();
312       }
313       //Sobald Drucktaster erneut bestaetigt: SD-Karte unmounten und
314       //aus dem Loop ueber Restart ausbrechen
315       else {
316          Init_success = 0;
317          esp_vfs_fat_sdmmc_unmount(); //SD-Karte unmounten
318
319          Init_success = 0;
320          digitalWrite(led1, LOW);
321          digitalWrite(led2, HIGH);
322          vTaskDelay(400 / portTICK_PERIOD_MS);
323          digitalWrite(led2, LOW);
324          vTaskDelay(50 / portTICK_PERIOD_MS);
325          ESP.restart();
326       }
327    }
```

Programmcode 3.8: Abfrage des Drucktasters; esp32_promiscuous.cpp (C++)

Über die Variable `buttonState` wird der Zustand des Drucktasters abgefragt. Nimmt dieser den Wert `HIGH` (gedrückt) an, wird der Wert der Variable `Init_success` geprüft. Ist dieser 0, wird das Programm initialisiert, ansonsten wird die Variable `Init_success` auf 0 gesetzt, die SD-Karte ungemountet und das erfolgreiche Beenden des Programmablaufs über `LED_READY` signalisiert. Anschließend wird der ESP32 neu gestartet.

```
329    //Wake-up fuer Powerbank, LED_PAKET wird als Signal genutzt ->
           alle 5 Sekunden triggern
330    else if (Init_success == 0) {
331       if (currentMillis - previousMillis >= 5000) {
332          previousMillis = currentMillis;
333          digitalWrite(powerbank, HIGH);
334          vTaskDelay(50 / portTICK_PERIOD_MS);
335          digitalWrite(powerbank, LOW);
336       }
337    }
```

Programmcode 3.9: Powerbank Wake-Up; esp32_promiscuous.cpp (C++)

Die zweite Aufgabe umfasst das Wake-Up Signal für eine Powerbank. Alle fünf Sekunden wird der GPIO, welcher der Variable `powerbank` zugeordnet ist, auf `HIGH` und nach 50 ms wieder auf `LOW` gesetzt. Hintergrundinformationen hierzu im Abschnitt 3.3.

```
339     //Kanalwechsel, wenn dieser true, Initialisierung erfolgreich
            und Switching Time erreicht ist
340     if (CHANNEL_SWITCHING && Init_success == 1 && (currentMillis -
            previousMillis >= SWITCHING_TIME / portTICK_PERIOD_MS)) {
341       previousMillis = currentMillis;
342       channel = (channel % 13) + 1;
343       esp_wifi_set_channel(channel, WIFI_SECOND_CHAN_NONE);
344     }
```

Programmcode 3.10: Automatischer Kanalwechsel; esp32_promiscuous.cpp (C++)

Die dritte Aufgabe umfasst den Kanalwechsel. Sofern `CHANNEL_SWITCHING` den Wert `true` besitzt, die Initialisierung erfolgreich abgeschlossen ist und der definierte Zeitintervall abgelaufen ist (`SWITCHING_TIME`), wird der Kanal inkrementiert und an `esp_wifi_set_channel(channel, WIFI_SECOND_CHAN_NONE);` übergeben.

3.2.3 Promiscuous Mode und Aufbereiten des Hexdumps

Die Funktion `void wifi_promiscuous(void* buffer, wifi_promiscuous_pkt_ type_t type)` wird durch den Callback (Abschnitt 3.2.2) aufgerufen. Dies geschieht bei jedem empfangenen Paket im Promiscuous Mode.

```
354     //Buffer parsen
355     wifi_promiscuous_pkt_t* packet = (wifi_promiscuous_pkt_t*)buffer;
356
357     /*Callback aktiv + Client Sniffing aktiviert = Deauth/4 way
            handshake sniffing -> nur Datenpakete
358     oder: Callback aktiv + Client Sniffing deaktiviert = alle Pakete
            sniffen*/
359     if ((client_sniffing == 1 && type == WIFI_PKT_DATA) ||
            client_sniffing == 0) {
360       //letzte 4 Bytes bei Management-Paketen abschneiden (Bug im
              ESP-IDF)
361       //if (type == WIFI_PKT_MGMT) (packet->rx_ctrl.sig_len) -= 4;
362       xRingbufferSend(packetRingbuf, packet,
              packet->rx_ctrl.sig_len, 1);
363     }
364 }
```

Programmcode 3.11: Promiscuous Mode; esp32_promiscuous.cpp (C++)

Der Buffer wird über `wifi_promiscuous_pkt_t* packet = (wifi_promiscuous _pkt_t*)buffer;` in ein geeignetes Format geparst, welcher in einem Ringpuffer abgelegt wird. Der Ringpuffer wird anschließend in der Funktion `void preparing_hexdump` verarbeitet, welche auf einem extra Kern ausgeführt wird. Mit diesem Verfahren kann der Ringpuffer verarbeitet, aufbereitet und auf der SD-Karte gespeichert werden, während im Hintergrund nach weiteren Paketen gesnifft wird. Ringpuffer funktionieren nach

dem First In – First Out Prinzip, das heißt alle Elemente, welche als erstes im Puffer abgelegt werden, werden auch als erstes dem Puffer entnommen. Der Ringpuffer besitzt eine feste Länge und überschreibt die ältesten Einträge zuerst sobald dieser voll ist.

```
128       uint8_t address1[12], address2[12], address3[12],
              address4[12];
129       for (int i = 0; i < 6; i++) {
130          address1[i] = { packet->payload[4 + i] };
131          address2[i] = { packet->payload[10 + i] };
132          address3[i] = { packet->payload[16 + i] };
133          address4[i] = { packet->payload[24 + i] };
134       }
135       //memcmp returned Wert 0 wenn mac_filter mit address1, -2, -3
              oder -4 bereinstimmt, andernfalls != 0
136       int a1, a2, a3, a4;
137       a1 = memcmp(mac_filter, address1, sizeof(mac_filter));
138       a2 = memcmp(mac_filter, address2, sizeof(mac_filter));
139       a3 = memcmp(mac_filter, address3, sizeof(mac_filter));
140       a4 = memcmp(mac_filter, address4, sizeof(mac_filter));
141
142       if ((a1 == 0) || (a2 == 0) || (a3 == 0) || (a4 == 0)) {
```

Programmcode 3.12: Filtern der Pakete nach mac_filter; esp32_promiscuous.cpp (C++)

Aus dem Puffer werden alle vier Adresse entnommen und in separate Variablen abgelegt. Es erfolgt ein Abgleich mit dem MAC-Filter. `memcmp` gibt den Wert 0 zurück, wenn einer dieser vier Adressen mit dem Filter übereinstimmt. Sollte der Filter leer sein, ist der Wert immer 0 (es wird also nicht gefiltert).

```
174             for (int i = 0; i < packet->rx_ctrl.sig_len; i++) {
175                if (i % 8 == 0) {
176                   //Offset hinzufuegen
177                   snprintf(hexbuf, sizeof(hexbuf), "%06X ", offset);
178                   strcat(hexbuf_dump, hexbuf);
179                   offset += 8;
180                }
181                //Hex anfuegen
182                snprintf(hexbuf, sizeof(hexbuf), "%02X ",
                      packet->payload[i]);
183                strcat(hexbuf_dump, hexbuf);
184                //Absatz, wenn: 8x Hex oder Frame zuende
185                if ((i % 8 == 7) || ((i + 1) ==
                      packet->rx_ctrl.sig_len )){
186                   snprintf(hexbuf, sizeof(hexbuf), "\n");
187                   strcat(hexbuf_dump, hexbuf);
188                }
189             }
```

Programmcode 3.13: Aufbereiten des Hexdumps; esp32_promiscuous.cpp (C++)

Der Dump wird in einen für Wireshark interpretierbares Format umkonvertiert. Am Anfang jeder Zeile wird der entsprechende Offset hinzugefügt, gefolgt vom Hexdump.

Die Länge des Dumps ist der Variable `rx_ctrl.sig_len` zu entnehmen. Über `snprintf` und `strcat` wird der Dump temporär als String in der Variable `hexbuf` bzw. `hexbuf_dump` abgelegt.

```
191             offset = 0;
192             //printf("%s", hexbuf_dump);
193
194             save_to_sdcard(hexbuf_dump);
195             memset(&hexbuf_dump[0], 0, sizeof(hexbuf_dump));
```

Programmcode 3.14: Speichern des Hexdumps auf SD-Karte; esp32_promiscuous.cpp (C++)

Sobald das Paket vollständig als Hexdump aufbereitet wurde, wird der Offset zurück auf 0 gesetzt und mittels der Funktion `save_to_sdcard(hexbuf_dump)` die Daten auf der SD-Karte gesichert. `memset` löscht danach den Inhalt der Variablen `hexbuf_dump`, sodass dieser nicht redundant gesichert wird.
`vRingbufferReturnItem(packetRingbuf, packet)` in Zeile 199 entfernt die Daten aus dem Ringpuffer.

```
93    //const char* data = (const char*) param;
94    sprintf(filenamebuf, "/sdcard/%s.pcap", filename);
95    //ESP_LOGI(TAG, "Opening file");
96    FILE* f = fopen(filenamebuf, "a");
```

Programmcode 3.15: Öffnen der Capture-Datei; esp32_promiscuous.cpp (C++)

Die Funktion `voidsave_to_sdcard` legt die Daten auf der SD-Karte ab. Der übergebene Dateiname aus der Variable `filename` wird die Dateiendung .pcap und der Pfad angefügt. Diese Information wird in der Variable `filenamebuf` abgelegt und diese anschließend mit dem Parameter „a“ (append = hinzufügen) geöffnet. Sollte es hier zu einem Fehler kommen, wird dies mit `LED_ERROR` signalisiert und der ESP32 neugestartet.

```
107   fwrite(data, sizeof(char), strlen(data), f);
108   fclose(f);
```

Programmcode 3.16: Schreiben der Daten auf SD-Karte; esp32_promiscuous.cpp (C++)

Andernfalls werden die Daten auf die SD-Karte geschrieben und die Datei anschließend geschlossen.

3.2.4 Deauthentification-Attack / Four-Way Handshake

Optional kann in Zeile 41 die Variable `#define` DEAUTHENTIFICATION auf `true` gesetzt werden. In diesem Fall wird in der Funktion `void` Initialisierung() in Zeile 305 die Funktion `void` deauthentification_attack() aufgerufen.
Es wird eine sogenannte „Deauthentification-Attack“ auf den Access Point ausgeführt (Schicht 2 im OSI-Referenzmodell). In 802.11 Netzwerken authentifizieren sich Clients gegenüber einem Access Point, bevor es zur eigentlichen Kommunikation kommt. Bestandteil des Authentifizierungs-System ist ebenfalls die Möglichkeit der Deauthentifizierung, sodass Client und Access Point die Kommunikation abbrechen und der Client das Netzwerk verlassen kann. Dies erfolgt jedoch nicht über einen verschlüsselten Weg, sodass Pakete dieser Art auch von außerhalb entweder an den Client oder den Access Point gesendet werden können, um die Verbindung abzubrechen [BS03].
Hintergrund der Deauthentification Attack in diesem Kontext ist das Erzwingen eines neuen Verbindungsaufbaus von Client und Access Point. Beim anschließenden Verbinden kommt es zum Four-way handshake, wobei die vier EAP-Key-Messages gesnifft werden können. Diese dienen u.a. Wireshark zum Entschlüsseln des gesamten Netzwerkverkehrs. Auch kann (z.B. mithilfe eines Brute-Force-Angriffs) daraus der Schlüssel gewonnen werden.

```
203 void deauthentification_attack()
204 {
205    client_sniffing = 1;
206    esp_wifi_set_promiscuous(true);
207    while (client_sniffing == 1) { //zufaelligen Client suchen
208         printf("suche mac...\n");
209    }
210    esp_wifi_set_promiscuous(false);
211    char* packet = Deauth.builder(mac_filter, client);
212
213    //WiFi Driver zum versenden initialisieren
214    esp_interface_t wifi_if;
215    void* wifi_eth = NULL;
216    wifi_if = tcpip_adapter_get_esp_if(wifi_eth);
217
218    for (int i = 0; i < 200; i++) {
219       //esp_wifi_internal_tx(wifi_if, (void*)packet,
             sizeof(packet));
220       int len = sizeof(packet);
221       esp_err_t esp_wifi_80211_tx(wifi_interface_t ifx, const void
             *packet, int len);
222       ESP_LOGI(TAG, "send packet");
223       vTaskDelay(100 / portTICK_RATE_MS);
224    }
225    vTaskDelay(10000 / portTICK_RATE_MS);
226    Init_success = 1;
227 }
```

Programmcode 3.17: deauthentification_attack; esp32_promiscuous.cpp (C++)

Die Variable `Client_sniffing` nimmt den Wert 1 an und der Promiscuous Mode wird aktiviert. Dies dient dem Filtern nach Paketen vom Typ Data:

```
359    if ((client_sniffing == 1 && type == WIFI_PKT_DATA) ||
           client_sniffing == 0) {
```

Programmcode 3.18: Data Paket Filter - esp32_promiscuous.cpp (C++)

Ebenfalls wird in der Funktion `void preparing_hexdump` die MAC-Adresse eines Clients gefiltert:

```
142        if ((a1 == 0) || (a2 == 0) || (a3 == 0) || (a4 == 0)) {
143           digitalWrite(led3, HIGH);
144           //DEAUTHENTIFICATION true: Lege Adresse 1 oder 3 in
                 client[] ab und beende Funktion an dieser Stelle
145           if (client_sniffing == 1) {
146              //Abfrage des DS Status um herauszufinden, welche
                    Adresse der Client ist
147              //(siehe Tabelle 2.2: Erlaeuterung des Distribution
                    Systems)
148              sprintf(buf, "%02X\n", packet->payload[1]);
149              char* ds_status = HexToBinary.convert(buf);
150
151              //Parsen des DS in int -> To DS bei [8], From DS bei [7]
152              int from_ds = ds_status[8] - '0';
153              int to_ds = ds_status[7] - '0';
154
155              if ((from_ds == 0 && to_ds == 0) || (from_ds == 0 &&
                    to_ds == 1)) {
156                 //Adresse 1 ist Client
157                 for (int i = 0; i < 6; i++) {
158                    client[i] = packet->payload[4 + i];
159                 }
160              }
161              else if ((from_ds == 1 && to_ds == 0) || (from_ds == 1
                    && to_ds == 1)) {
162                 //Adresse 3 ist Client
163                 for (int i = 0; i < 6; i++) {
164                    client[i] = packet->payload[16 + i];
165                    client_sniffing = 0;
166                 }
```

Programmcode 3.19: Client sniffing; esp32_promiscuous.cpp (C++)

Über den DS Status wird ermittelt, welche der vier Adressen der MAC-Adresse des Clients entspricht. Der DS Status befindet sich im ersten Index des Payloads. Anschließend wird mittels der Headerdatei `HexToBinary` (siehe Anhang B.2) der DS Status in eine Binärzahl umkonvertiert und durch die Kombination aus `from_ds` und `to_ds` die MAC-Adresse ermittelt (siehe Tabelle 2.2), welche in der Variable `client[i]` abgelegt wird. Danach wird die Variable `Client_sniffing` wieder auf 0 gesetzt. Die `while`-Schleife wird dadurch verlassen und der Promiscuous Mode deaktiviert. Die Headerdatei `Deauth`

setzt sich zusammen mit der MAC-Adresse des Access Points und des Clients ein Deauthentification Paket zusammen (siehe Anhang B.3). Dies entspricht vom Aufbau:

```
16      0xC0, 0x00, //type und subtype (c0 -> deauth)
17      0x00, 0x00, //Duration
18      0xBB, 0xBB, 0xBB, 0xBB, 0xBB, 0xBB, //Ziel (Client)
19      0xCC, 0xCC, 0xCC, 0xCC, 0xCC, 0xCC, //Absender (AP)
20      0xCC, 0xCC, 0xCC, 0xCC, 0xCC, 0xCC, //BSSID (AP)
21      0x00, 0x00, //Fragment und Sequenz
22      0x01, 0x00 //reason code (1 = unspecified reason)
```

Programmcode 3.20: Aufbau Deauthentification Paket; Deauth.cpp (C++)

Zeile 18 und 19 werden entsprechend mit den MAC-Adressen von Access Point und Client ersetzt. Anschließend wird der Wi-Fi Adapter aktiviert und die Pakete versendet.

Derzeit ist das Versenden von Deauthentification Paketen mit dem ESP32 nicht möglich (es können zwar Management Pakete versendet werden, jedoch bislang nur Beacon Frames), da die Funktion noch nicht in der `esp_wifi.h` deklariert ist[9]. Daher fehlt in dieser Funktion noch das anschließende sniffen des Four-way handshakes bzw. ein Filtern der vier EAP-Key-Messages.

Eine mögliche Alternative besteht in der Kombination eines ESP32 und eines ESP8266. Ein potenzieller Schaltplan ist der Abbildung 3.4 zu entnehmen. Der ESP82666 (in der SDK Version 1.3) ist durch eine Konfiguration der Datei `user_interface.h` fähig Pakete zu versenden. Offiziell wurde diese Funktion mit der Einschränkung nur Beacon Frames zu verschicken in der SDK Version 1.4 implementiert, jedoch gab es bereits in der Version 1.3 diese Funktion ohne solche Einschränkungen. Dafür müssen lediglich die Headerfiles angepasst werden. Die Konfiguration umfasst somit das Hinzufügen folgender Zeilen in der Datei `user_interface.h` vor `#endif`:

```
typedef void (*freedom_outside_cb_t)(uint8 status);
int wifi_register_send_pkt_freedom_cb(freedom_outside_cb_t cb);
void wifi_unregister_send_pkt_freedom_cb(void);
int wifi_send_pkt_freedom(uint8 *buf, int len, bool sys_seq);
```

Nach dem Hinzufügen dieser Zeilen ist es möglich über den Befehl `wifi_send_pkt_freedom()` Datenpakete zu versenden. Während somit der ESP8266 Deauthentification Pakete verschickt, befindet sich der ESP32 im Promiscuous Modus und zeichnet den Handshake auf.
Über die Pins RX und TX können beide MCUs miteinander kommunizieren. So ist es möglich dem ESP8266 mitzuteilen, welche MAC-Adresse jeweils der Access Point und der dazugehörige Client besitzt, um das Paket entsprechend aufzubauen. Während der ESP8266 Deauthentification-Pakete versendet, um Client und Access Point voneinander zu trennen und damit einen neuen Verbindungsaufbau erzwingt, befindet sich der ESP32 im Promiscous Mode und zeichnet indes den Four-Way Handshake auf.

[9] https://github.com/espressif/esp-idf/issues/677#issuecomment-316313610, letzter Zugriff: 27.08.2017

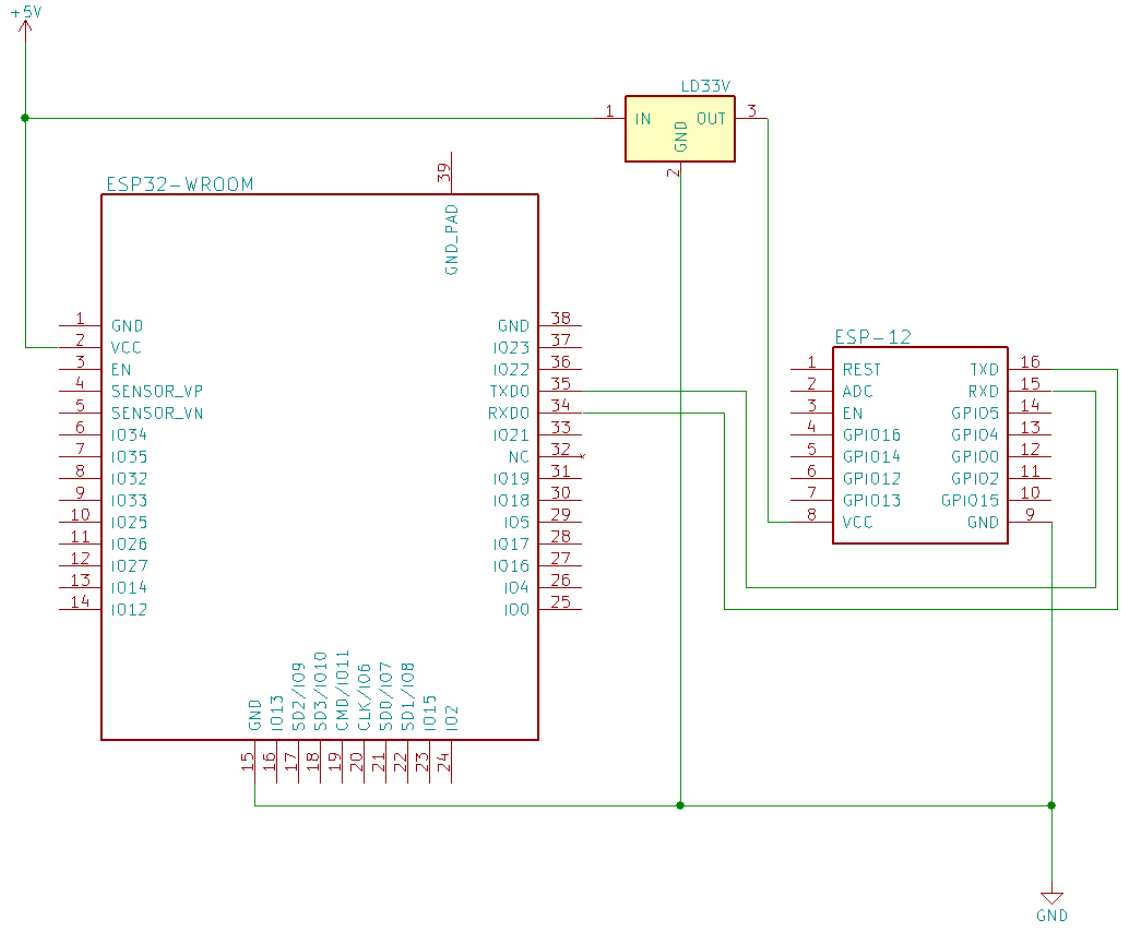


Abbildung 3.4: Schaltplan: Kommunikation zwischen ESP32 und ESP8266

3.3 Stromverbrauch und -versorgung

Die benötigte Stromstärke dieses embedded Device liegt im Ruhezustand (ESP32 befindet sich an der Spannungsversorgung, Programm wird jedoch nicht über den Drucktaster gestartet) bei circa 0.06A. Im Betrieb steigt der Verbrauch auf etwa 0,18A an. Gemessen wurde dies mit einem USB-Multimeter, welches eine Toleranz von $\pm 1\%$ aufweist. Die 5V Spannung, welche für den Betrieb notwendig sind, bezieht der ESP32 über eine Micro-USB-Schnittstelle.

Bei der Verwendung mit einer Powerbank ist darauf zu achten, dass der minimale Ladestrom, der dem Datenblatt entnommen oder beim Hersteller angefragt werden kann, nicht unterschritten wird, da ansonsten ein Großteil der heutigen Powerbanks automatisch nach einem definierten Zeitintervall abschalten. Dies hat den Hintergrund, dass die Powerbank das angeschlossene Gerät als „geladen betrachtet".
Dies kann beispielsweise mittels eines Transistors umgangen werden, indem der aktuelle Ladestrom kurzzeitig über den Sollwert gepulst wird.
Dieser Transistor funktioniert wie ein Schalter: Strom fließt vom Collector zum Emitter, wenn an der Basis eine Spannung anliegt.

Am Beispiel der „VOLTCRAFT PB-17 Li-Ion 10400 mAh"-Powerbank liegt der minimale Ladestrom bei 100mA (nach Anfrage beim Hersteller). Dies sei I_c = 100mA. Abbildung 3.5 zeigt einen solchen Schaltplan.

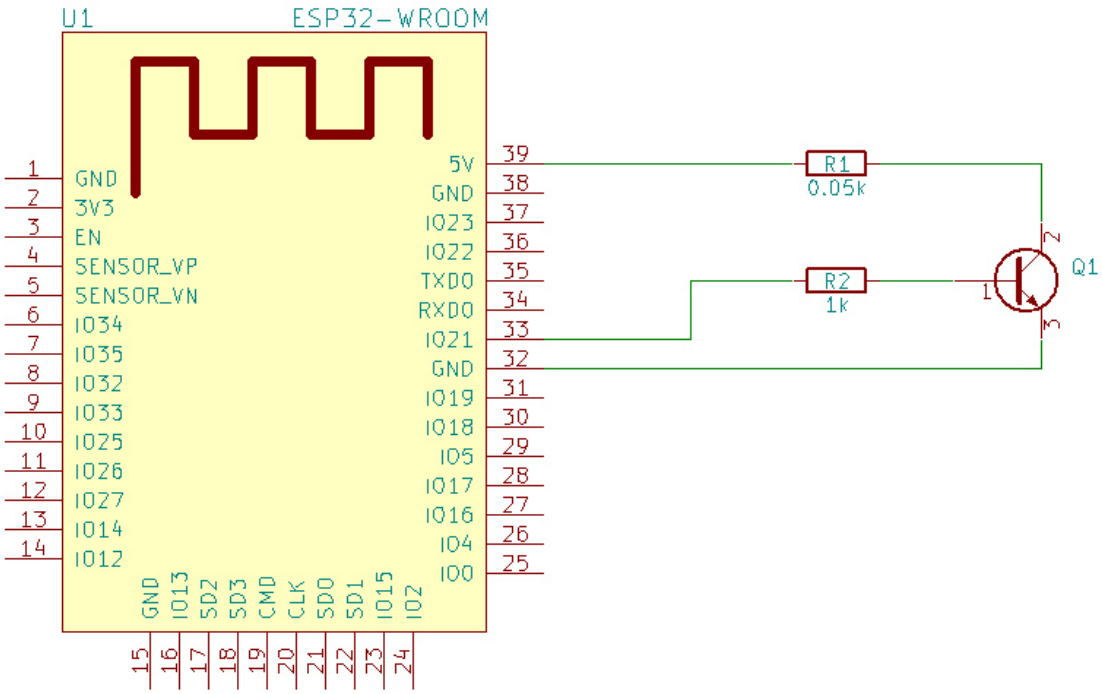


Abbildung 3.5: Schaltplan: Powerbank Wake-Up Signal mittels Transistor

Der benötigte Widerstand zwischen Collector (5V) und Emitter berechnet sich durch:

$$R = \frac{U}{I} = \frac{5V}{100mA} = 50\Omega$$

In dieser Schaltung wird der BC547B NPN Transistor genutzt, welcher mindestens 0.7V an der Basis zum Schalten benötigt. Der für den maximalen Kollektorstrom nötige sättigende Schaltbetrieb wird mit $V_{CE(sat)}$ angegeben. Die Stromverstärkung liegt laut Datenblatt bei minimal 10 [SPC08]. Die Ausgangsspannung ergibt sich abzüglich der Basis-Emitterspannung des Transistors von 0.7V, welche in der Kalkulation berücksichtigt werden muss [SZ14, S. 182]. Weiterhin ist bei der Verwendung mit anderen Transistoren darauf zu achten, dass die maximale Stromstärke eines GPIOs vom ESP32 im Datenblatt mit 40mA angegeben wird [Esp17d, S. 28].

Daraus berechnet sich der Basisstrom I_b wie folgt:

$$I_b = \frac{I_c}{V_{CE(sat)}} = \frac{100mA}{10} = 10mA$$

Damit lässt sich der Widerstand R_b an der Basis folgendermaßen ableiten:

$$R_b = \frac{U_E - 0,7V}{I_b} = \frac{3,3V - 0,7V}{10mA} = 260\Omega$$

Diese Schaltung ist notwendig, falls das embedded Device noch eine kurze Zeit im Ruhezustand verharren soll, bevor das Programm gestartet wird. Sie ist daher optional, da die Ladelast im Betrieb bei >100mA liegt. Alternativ kommt diese Schaltung auch zum Einsatz, falls die genutzte Powerbank eine höhere Ladelast erfordert, als die vom embedded Device benötigten 180mA.
Wahlweise können auch direkt Lithium-Polymer-Akkus als Stromversorgung genutzt werden. Dies hat den Vorteil, dass auf die interne Elektronik einer Powerbank und die daraus resultierende automatische Abschaltung verzichtet werden kann.

Bei der Verwendung von Schaltnetzteilen ist auf mögliche HF-Störungen und die elektromagnetische Verträglichkeit gegenüber dem ESP32 zu achten. Schaltnetzteile arbeiten mit einer Arbeitsfrequenz um 100 kHz [Hus10, S.82] und können ungewollt den Empfang des Wi-Fi Chips stören, was im Einzelfall zu überprüfen ist.

4 Ergebnisse

4.1 Mitschnitt

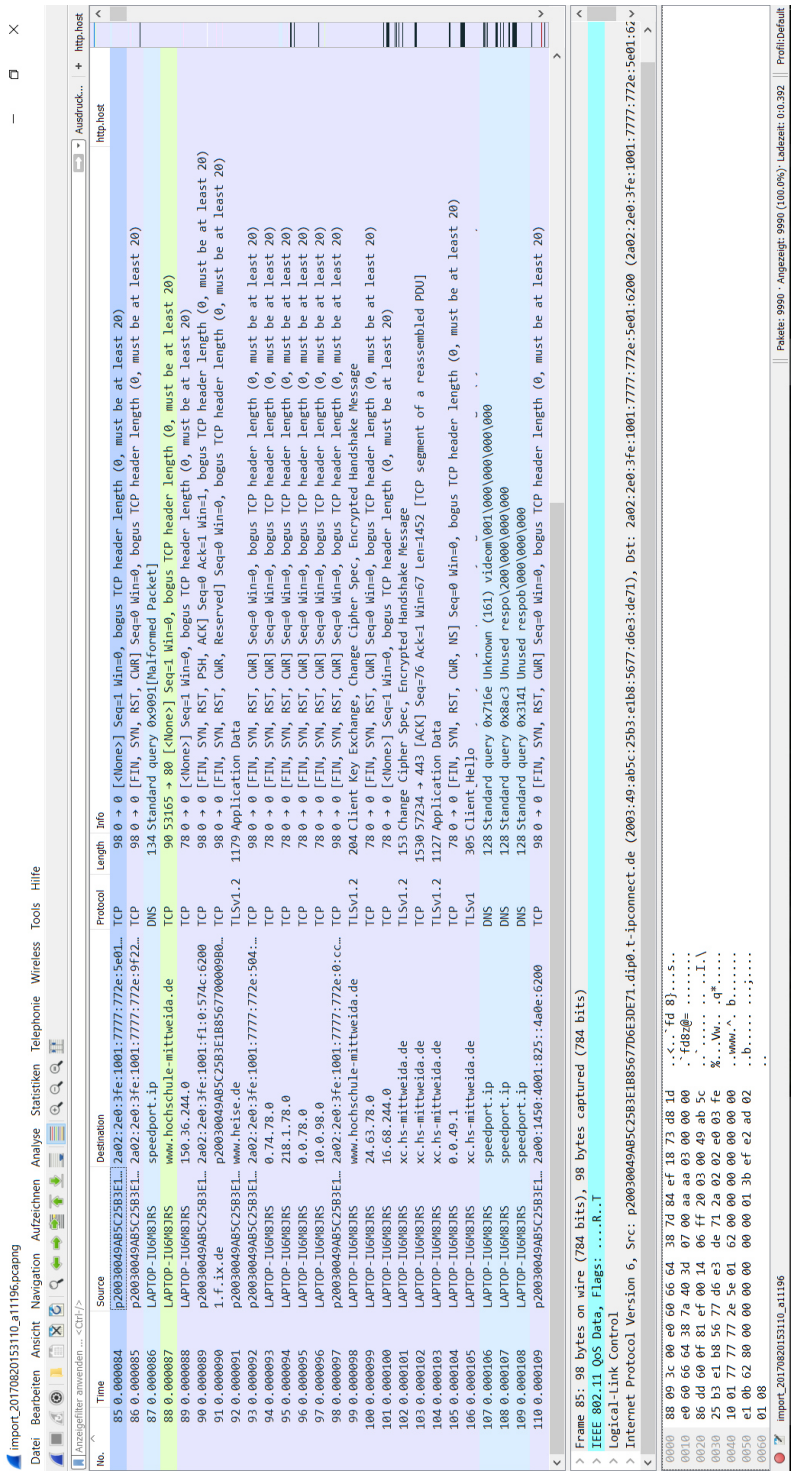


Abbildung 4.1: Entschlüsselter Datenverkehr in Wireshark

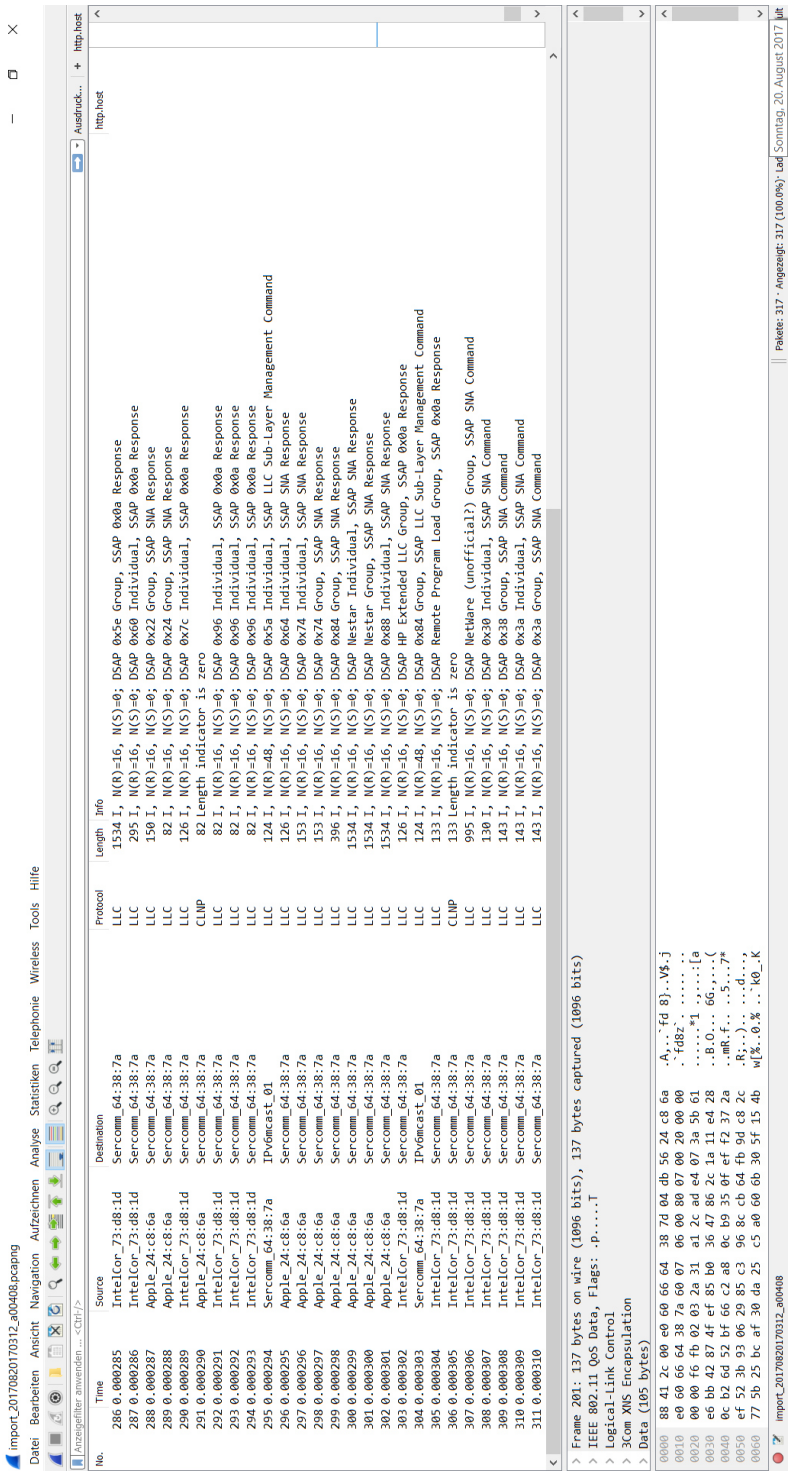


Abbildung 4.2: Verschlüsselter Datenverkehr in Wireshark

4.2 Benchmark

4.2.1 Embedded Device

Für den Benchmark wird das in Python geschriebene Werkzeug „Scapy“ genutzt, welches eine Bibliothek zur Paketmanipulation darstellt. Mithilfe von „Scapy“ lassen sich somit Pakete erstellen und versenden.

```
12 #Pakete versenden
13 for i in range(0,1000):
14    #time.sleep(0.1)
```

Programmcode 4.1: Anzahl der Pakete definieren; scapy_beaconsender.py (Python)

Das Programm „scapy_beaconsender.py“ (modifizierte Variante des Programms „newSSID.py“ aus dem Artikel „Forging WiFi Beacon Frames Using Scapy“ des Autors William Hurer-Mackay [HM16]) versendet 100, 1.000 und anschließend 10.000 Beacon Frames (á 79 Bytes). Jeder davon nach einem definierten Zeitabstand. Dieser startet bei 0ms (sleep ist auskommentiert) und wird schrittweise inkrementiert, bis das embedded Device 100% der Pakete empfängt. Jede Messung wird dreimal wiederholt um ein repräsentatives Ergebnis zu erhalten.

Es wird hiernach überprüft, wie viele der Beacon Frames vom EPS32 aufgezeichnet werden konnten und dies in Diagrammen gegenübergestellt. Anschließend wird aus der benötigten Zeit die Geschwindigkeit in Mbps über das arithmetische Mittel ermittelt.

Zum Versenden der Pakete wird der Alfa AWUS036NH Netzwerkadapter genutzt, welcher vorher in den Monitor Mode versetzt werden muss. Eine Möglichkeit ist die Verwendung des Programms „airmon-ng“ über folgendem Befehl:

`airmon-ng start wlan0 6`

In Linux kann mittels `iwconfig` der Name des Interface vorher im Terminal überprüft werden, sodass dieser korrekt übergeben wird. Die Zahl nach dem Interface (`6`) bestimmt den Kanal, auf welchem die Pakete versendet werden.

```
16    dot11 = Dot11(type=0, subtype=8, addr1=broadcast, addr2=bssid,
          addr3=bssid)
17    beacon = Dot11Beacon(cap='ESS')
18    essid = Dot11Elt(ID='SSID',info=Beacon_SSID,
          len=len(Beacon_SSID))
19    rsn = Dot11Elt(ID='RSNinfo', info=(
20    '\x01\x00' #RSN Version 1
21    '\x00\x0f\xac\x02' #Group Cipher Suite : 00-0f-ac TKIP
22    '\x02\x00' #2 Pairwise Cipher Suites (next two lines)
23    '\x00\x0f\xac\x04' #AES Cipher
24    '\x00\x0f\xac\x02' #TKIP Cipher
25    '\x01\x00' #1 Authentication Key Managment Suite
26    '\x00\x0f\xac\x02' #Pre-Shared Key
27    '\x00\x00')) #RSN Capabilities (no extra capabilities)
```

Programmcode 4.2: Aufbau des Beacon Frames; scapy_beaconsender.py (Python)

Innerhalb dieser Umgebung wird der Beacon Frame definiert, welcher als Type 0 (Management Frame) und Subtype 8 (Beacon Frame) besitzt:

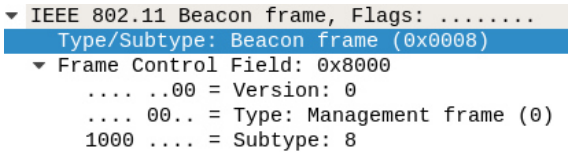


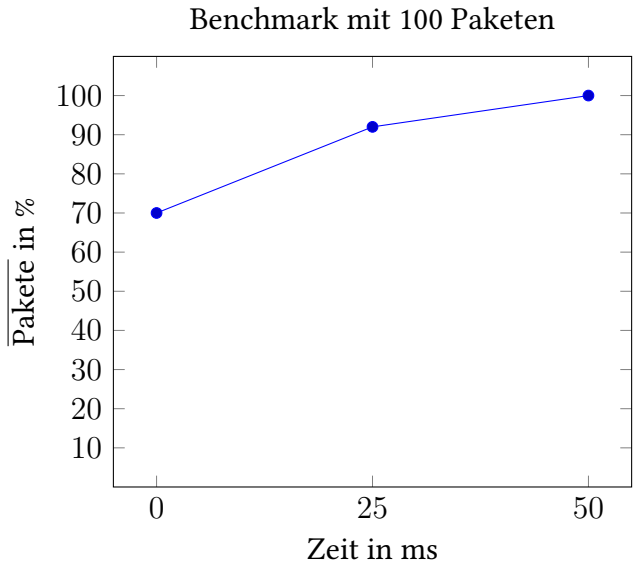
Abbildung 4.3: Type/Subtype des Beacon Frames in Wireshark

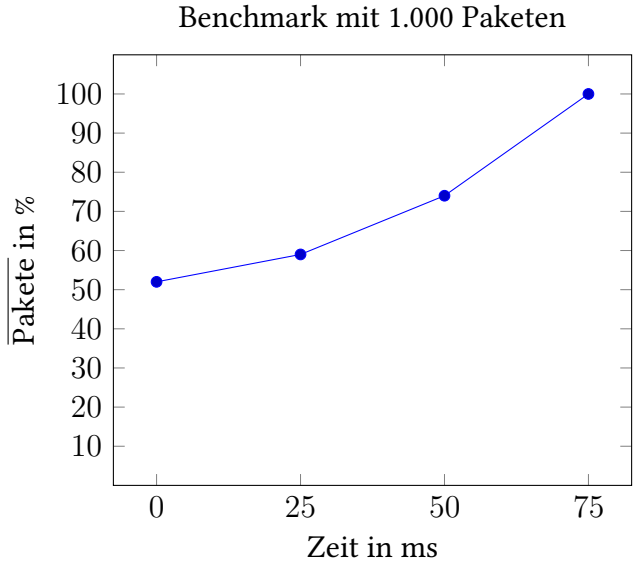
`addr1` entspricht der Destination MAC Adresse, `addr2` der Source MAC Adresse und `addr3` der MAC-Adresse des Access Points.
Die Variable `beacon` bestimmt die Eigenschaften des Access Points: Dieser wird als ESS-Netzwerk. `Dot11beacon` ist der Konstruktor um in der Umgebung einen 802.11 Beacon Frame zu definieren. Die Variable `RSN` definiert das Netzwerk als WPA2. Zeile 21 bestimmt die RSN Version (hier 1)und Zeile 22 - 23 die Cipher Suite (kryptographisches Verfahren, hier AES und TKIP).

```
6  #Settings
7  Beacon_SSID = 'Fake_Beacon'    #network name
8  interface = 'wlan0mon'    #interface
9  bssid = "e1:e1:e1:e1:e1:e1"    #broadcast
10 broadcast = "f2:f2:f2:f2:f2:f2"  #bssid
```

Programmcode 4.3: Definieren des Pakets; scapy_beaconsender.py (Python)

`Beacon_SSID` trägt den Namen des Beacon Frames. Das Interface, welches zum Versenden der Pakete genutzt wird, wird über `interface` angegeben.
`bssid` entspricht `addr2` und `addr3`, `broadcast` `addr1`.


Benchmark mit 100 Paketen
Pakete in %
100
90
80
70
60
50
40
30
20
10
0
25
50
Zeit in ms


Benchmark mit 1.000 Paketen
Pakete in %
100
90
80
70
60
50
40
30
20
10
0
25
50
75
Zeit in ms

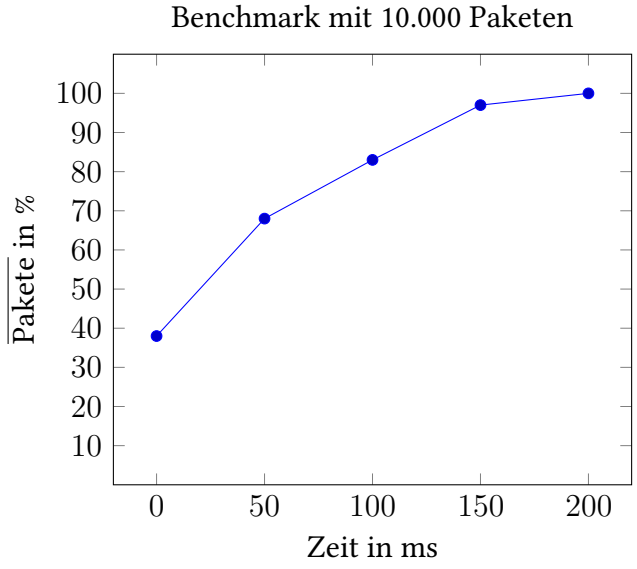

Benchmark mit 10.000 Paketen
Pakete in %
100
90
80
70
60
50
40
30
20
10
0
50
100
150
200
Zeit in ms

Tabelle 4.1: Ergebnis des embedded Device Benchmarks

Versendete Pakete	Messung	Zeitabstand in ms	Pakete	$\overline{\text{Pakete}}$	Größe in Byte	Zeit in Sekunden	Mbps	$\overline{\text{Mbps}}$
100	1	0	67	70	5293	5,48	0,007727	0,006976
	2		54		4266	5,2	0,006563	
	3		90		7110	6,78	0,008389	
	1	25	85	92	6715	8,19	0,006559	
	2		100		7900	8,34	0,007578	
	3		92		7268	7,61	0,007640	
	1	50	100	100	7900	10,46	0,006042	
	2		100		7900	10,23	0,006178	
	3		100		7900	10,35	0,006106	
1.000	1	0	518	521	40922	49,13	0,006663	0,005264
	2		502		39658	48,56	0,006533	
	3		542		42818	51,23	0,006686	
	1	25	582	589	45978	80,55	0,004566	
	2		601		47479	82,71	0,004592	
	3		585		46215	80,9	0,004570	
	1	50	764	740	60356	98,32	0,004911	
	2		741		58539	97,94	0,004782	
	3		716		56564	93,87	0,004821	
	1	75	1000	1000	79000	127,98	0,004938	
	2		1000		79000	125,67	0,005029	
	3		1000		79000	124,46	0,005078	
10.000	1	0	3811	3795	301069	490,18	0,004914	0,003614
	2		3790		299410	494,6	0,004843	
	3		3784		298936	493,85	0,004843	
	1	50	6807	6842	537753	1046,63	0,004110	
	2		6848		540992	1049,32	0,004125	
	3		6871		542809	1152,32	0,003768	
	1	100	8357	8346	660203	1448,17	0,003647	
	2		8388		662652	1450,32	0,003655	
	3		8294		655226	1450,88	0,003613	
	1	150	9702	9717	766458	2039,81	0,003006	
	2		9740		769460	2042,23	0,003014	
	3		9708		766932	2040,01	0,003008	
	1	200	10000	10000	790000	2485,94	0,002542	
	2		10000		790000	2492,32	0,002536	
	3		10000		790000	2439,7	0,002590	

Die Geschwindigkeit in bps wird berechnet durch die Gesamtgröße aller empfangenen Pakete in Byte multipliziert mit 8 (1 Byte = 8 Bit) und dem anschließenden Dividieren durch die benötigte Zeit (in Sekunden). Daraus lässt sich die Geschwindigkeit in Mbps ableiten, indem bps durch 1.000.000 dividiert wird:

$$\frac{(\text{Größe in Byte} \cdot 8)}{\text{Zeit (t) in Sekunden}} = bps \quad \Rightarrow \quad \frac{\text{bps}}{1.000.000} = Mbps$$

Am Beispiel anhand von 100 versendeten und davon 67 empfangenen Paketen in 5,48 Sekunden:

$$\frac{\left(\frac{5293\ \text{Byte} \cdot 8}{5,48\ \text{Sekunden}}\right)}{1.000.000} = \frac{7727\ \text{bps}}{1.000.000} = \underline{\underline{0,007727\ \text{Mbps}}}$$

Die durchschnittliche Geschwindigkeit in Mbps berechnet sich durch das arithmetische Mittel von $\overline{\text{Mbps}}$:

$$\begin{aligned}\overline{\text{Mbps}_{Total}} &= \frac{1}{n}\sum_{i=1}^{n}\text{Mbps}_i \\ &= \frac{1}{3}(0,006976\ \text{Mbps} + 0,005264\ \text{Mbps} + 0,003614\ \text{Mbps}) \\ &= \underline{\underline{0,005284\ \text{Mbps}}}\end{aligned}$$

Dies entspricht

$$\frac{0,005284\ \text{Mbps}}{8} = 0,0006605\ \text{MB/s} \cdot 1024 = \underline{\underline{0,676\ \text{kB/s}}}$$

In einer Stichprobe wurden 30 Minuten lang Pakete ohne jeglichen Zeitabstand versendet und anschließend die Anzahl an aufgezeichneten Pakete überprüft. Dabei konnte festgestellt werden, dass eine Gesamtpaketgröße von 911646 Bytes aufgezeichnet werden konnte. Dies entspricht:

$$\frac{\left(\frac{911646\ \text{Byte} \cdot 8}{1800\ \text{Sekunden}}\right)}{1.000.000} = \frac{4051,76\ \text{bps}}{1.000.000} = \underline{\underline{0,00405176\ \text{Mbps}}}$$

Zum Überprüfen dieses Ergebnisses wird die Standardabweichung herangezogen. Die empirische Varianz s^2 von Mbps beträgt:

$$\begin{aligned}s^2(\text{Mbps}) &= \frac{1}{n}\sum_{i=1}^{n}(\text{Mbps}_i - \overline{\text{Mbps}_{total}})^2 \\ &= \frac{1}{3}(0,006976 - 0,005284)^2 + (0,005264 - 0,005284)^2 + (0,003614 - 0,005284)^2 \\ &= \underline{\underline{0{,}0000028260813}}\end{aligned}$$

Um die Standardabweichung s abzuleiten, wird aus der empirischen Varianz s^2 die (quadratische) Wurzel gezogen:

$$\begin{aligned} s(\text{Mbps}) &= \sqrt{s^2(\text{Mbps})} \\ &= \sqrt{0,0000028260813} \\ &= \underline{\underline{0,0016811}} \end{aligned}$$

Die ermittelte Geschwindigkeit 0,00405176 Mbps der Stichprobe weicht vom Mittelwert 0,005284 Mbps ab, aber aufgrund der Standardabweichung von $\pm$0,0016811 Mbps liegt dies im Rahmen aller möglichen Ergebnisse.

4.2.2 SD-Karte

Für den SD-Karten Benchmark gibt es innerhalb des ESP-IDFs eine Funktion, welche mittels

```
cd $IDF_PATH/tools/unit-test-app/
make flash monitor TEST_COMPONENTS=sdmmc ESPPORT=/dev/ttyUSB0}
```

aufgerufen werden kann. `/dev/ttyUSB0` ist entsprechend der genutzten seriellen Schnittstelle zu ersetzen.

```
Name: SD
Type: SDHC/SDXC
Speed: default speed
Size: 7691MB
CSD: ver=1, sector_size=512, capacity=15751168 read_bl_len=9
SCR: sd_spec=2, bus_width=5
  sector   | count | size(kB)  | wr_time(ms) | wr_speed(MB/s)  |  rd_time(ms)  | rd_speed(MB/s)
        0 |     1 |      0.5  |      2.38   |       0.21      |      0.56     |      0.87
        0 |     4 |      2.0  |      2.76   |       0.71      |      1.33     |      1.47
        1 |    16 |      8.0  |      5.09   |       1.53      |      3.79     |      2.06
       16 |    32 |     16.0  |      8.94   |       1.75      |      7.09     |      2.21
       48 |    64 |     32.0  |     15.58   |       2.01      |     13.66     |      2.29
      128 |   128 |     64.0  |     27.73   |       2.25      |     26.87     |      2.33
 15751104 |    32 |     16.0  |      7.87   |       1.98      |      7.09     |      2.20
 15751104 |    64 |     32.0  |     14.44   |       2.16      |     13.67     |      2.29
 15751160 |     1 |      0.5  |      2.30   |       0.21      |      0.58     |      0.84
  7875584 |     1 |      0.5  |      2.29   |       0.21      |      0.58     |      0.84
  7875584 |     4 |      2.0  |      2.76   |       0.71      |      1.33     |      1.46
  7875584 |     8 |      4.0  |      3.51   |       1.11      |      2.16     |      1.81
  7875584 |    16 |      8.0  |      5.25   |       1.49      |      3.81     |      2.05
  7875584 |    32 |     16.0  |      7.79   |       2.01      |      7.09     |      2.20
  7875584 |    64 |     32.0  |     14.44   |       2.16      |     13.67     |      2.29
  7875584 |   128 |     64.0  |     41.28   |       1.51      |     26.88     |      2.33
  7875584 |     1 |      0.5  |      2.18   |       0.22      |      0.59     |      0.83
  7875584 |     8 |      4.0  |     13.29   |       0.29      |      4.57     |      0.86
  7875584 |   128 |     64.0  |    213.76   |       0.29      |     73.11     |      0.85
```

Bei diesem Benchmark wird in verschiedenen Sektoren eine bestimmte Anzahl an Blöcken geschrieben. Aus der Anzahl (jeder 0,5 kB), sowie der benötigten Zeit lässt sich die Lese- und Schreibgeschwindigkeit (MB/s) ableiten. Am Beispiel der Schreibgeschwindigkeit im Sektor 7875584 bei 0,5kB Blockgröße:

$$\frac{0,5 \text{ kB}}{1024} = 0,00048828125 \text{ MB}$$

$$\frac{0,00048828125 \text{ MB}}{0,00218 \text{ s}} = 0,22 \text{ MB/s}$$

Die durchschnittliche Schreibgeschwindigkeit berechnet sich auch hier durch das airthmetische Mittel aller Messungen:

$$\begin{aligned}\overline{\text{wr\_speed(MB/s)}} &= \frac{1}{19}\sum_{i=1}^{19} wr\_speed(MB/s)_i \\ &= \underline{\underline{1,2 \text{ MB/s}}}\end{aligned}$$

Das Ergebnis des Benchmarks variiert u.a. aufgrund des verwendeten SD-Karten Adapters, der Micro-SD-Karte und der Leitungslänge zwischen ESP32 und SD-Karten Adapter. Aus dem Benchmark des embedded Device ist zu entnehmen, dass die Geschwindigkeit bei 0,676 kB/s liegt. Damit ist die Geschwindigkeit (1,2 MB/s) der SD-Karte im 1-line Mode (default Speed) ausreichenend zur Verarbeitung.

5 Diskussion

5.1 Vergleich mit anderen Studien

In Bezug auf die vorliegende Arbeit sind folgende Veröffentlichungen relevant, auf die im Folgenden eingegangen wird:

a) „Discovering Human Presence Activities with Smartphones Using Nonintrusive Wi-Fi Sniffer Sensors: The Big Data Prospective“ von *Qin et al.* (2013) [QZLS13]
b) „Indoor Occupancy Tracking in Smart Buildings Using Passive Sniffing of Probe Requests“ von *Vattapparamban et al.* (2016) [Edw16]
c) „Real Time Wireless Packet Monitoring with Raspberry Pi Sniffer“ von *Turk et al.* (2014) [TDG14]
d) „Localization of WiFi Devices Using Probe Requests Captured at Unmanned Aerial Vehicles “ von *Acuna et al. (2017)* [Acu17]

Qin et al. (2013) erläutern in ihrer Veröffentlichung Ansätze zum sniffen von Paketen bei Smartphones. Das Ziel dieser Arbeit war das Aufspüren menschlicher Aktivitäten bezugnehmend auf Smartphones in der unmittelbaren Umgebungen ohne das Installieren von Apps und ohne das Stören des Netzwerkverkehrs. Dabei wird die Tatsache genutzt, dass Smartphones auch dann WLAN Traffic in Form von Probe-Request-Paketen erzeugen, auch wenn diese nicht mit einem Access Point verbunden sind. Bei einem Probe-Request fragt der Client nach verfügbaren Netzen in seiner Umgebung bzw. nach welchen mit denen er bereits verbunden war. Verfügbare Netze antworten mit einem Probe-Response-Paket. Diese Pakete gehören zu den Management Frames. Zur Realisierung dieses Projekts entwickelten *Qin et al.* ein embedded Device (auf ARM-Architektur), welches Probe-Requests von umliegenden Smartphones verarbeitet, den Kanal wechseln kann und die gesnifften Daten in einer Cloud (RESTful API) ablegt. Weiterhin werden redundante und non-mobile Daten gefiltert. Dieses System wurde ebenfalls mit einem Raspberry Pi und einem Back-End[12] Server getestet. Der Programmablauf wird auf drei wesentliche Schritte heruntergebrochen:

- Daten-gewinnende-Phase: In der ersten Phase werden soviele Daten wie nur möglich aufgezeichnet. Wichtig dafür ist der interne Algorithmus und die korrekte Wahl des Kanals.
- Daten-Speicher-Phase: Diese Phase beschäftigt sich mit dem Herausfiltern redundanter Daten.
- Daten-Analyse-Phase: In der letzten Phase werden wichtige Informationen aus den Daten gewonnen: Dies betrifft die Signalstärke, Source MAC Adresse, Desti-

[12] Teil eines IT-Systems, welcher sich mit der Datenverarbeitung im Hintergrund beschäftigt

den Daten gewonnen: Dies betrifft die Signalstärke, Source MAC Adresse, Destination MAC Adresse, Probe/Data Typ und Zeitstempel.

Qin et al. beschreiben das statische Zuweisen eines Kanals als „Fixed Channel Allocation (FCA)“ und stellt im weiteren drei Alternativen dazu vor:

- Average Time Channel Allucation (ATCA): Kanal wird autark in einem bestimmten zeitlichen Abstand gewechselt.
- Channel Activeness Based Channel Fair Allocation (CACFA): Mit dieser Methode wird 60 Minuten lang Datenverkehr aufgezeichnet. In den ersten 5 Minuten wird die Anzahl der Pakete je Kanal gezählt und anschließend die Kanäle in absteigender Reihenfolge nach der Anzahl der Pakete sortiert. Die übrigen 55 Minuten Laufzeit werden in 15-Sekunden-Intervalle aufgeteilt. Jedem Kanal werden diese Intervalle in Abhängigkeit der erhaltenen Pakete zugeteilt. Jeder Kanal erhält mindestens einen 15-sekündigen Zeitintervall.
- Channel Activeness Based Optimized Channel Allocation (CAOCA): Dieser Algorithmus baut auf CACFA auf. Kanäle mit wenig Traffic werden jedoch nicht beachtet. Die Zuteilung der 15-Sekunden-Intervalle stoppt hier, sobald 95 Prozent aller Intervalle aufgeteilt wurden.

In einem einmonatigen Versuch innerhalb eines Gebäudes konnte festgestellt werden, dass das CACFA- und CAOCA-Verfahren 3,8mal präziser ist, als der am wenigsten genutzte Kanal und 7 Prozent mal präziser, als der am Häufigsten genutzte Kanal im FCA-Verfahren [QZLS13].

Auf die vorliegende Arbeit sind die Ergebnisse von *Qin et al.* im Bereich der effektiven Kanalwahl bedeutsam. Das embedded Device wechselt entweder automatisch den Kanal (hier ist eine Übergabe der Zeitdauer in Millisekunden, bis es zum Wechsel kommt, notwendig) oder aber es wird ein Kanal statisch übergeben. Die zweite Variante deckt sich mit dem von *Qin et al.* beschriebenem „FCA“-Vorgehen. Die Implementierung der von *Qin et al.* beschriebenen Verfahren des CACFA- oder des CAOCA-Verfahrens wäre ein wichtiger Schritt um effizienter einen Großteil des Netzwerkverkehrs aufzuzeichnen. Dazu könnte eine entsprechende Variable (bspw. ein Array) bei jedem empfangenen Paket inkrementiert werden, welcher dem aktuellen Kanal entspricht. Ein möglicher Programmausschnitt, um dies zu erreichen ist Programmcode 5.1 zu entnehmen. Durch die Gesamtanzahl aller Pakete lässt sich somit die relative Häufigkeit eines jeden Kanals bestimmen und damit ein Ranking erstellen. Aufgrund dieses Rankings wird entschieden, welcher Kanal wieviel „Sniffing-Zeit“ bekommt. Dieses Verfahren ist optimal geeignet, um die Performance des embedded Device zu steigern.

```
uint8_t channel; //Aktuell genutzter Kanal
char channel_buffer[256]; //Groesse des Buffers anpassen

void wifi_promiscuous(void* buffer, wifi_promiscuous_pkt_type_t
    type) {
  channel_buffer[channel]++;
  for (int i = 1; i <= 13; i++) {
    printf("%i ", channel_buffer[i]);
  }
}
```

Programmcode 5.1: Anzahl der Pakete je Channel zählen und Ausgabe dieser; Beispiel (C++)

Ein weiteres Beispiel befindet sich in der Veröffentlichung von *Vattapparamban et al.* (2016). Hier werden Möglichkeiten vorgestellt, um die Anwesenheit von Personen in Räumen festzustellen. Solche Systeme kommen vor allem in Smart Buildings zum Einsatz, bspw. um die Energieeffizienz zu erhöhen und um weniger Strom zu verbrauchen. Auch Unternehmen können solche Informationen für Marketing-Strategien nutzen, da aus den ermittelten Daten abgeleitet werden kann, wieviele potenzielle Käufer wann den Laden betreten. Die bisherigen Varianten verlangen die Installation einer großen Anzahl von Monitoring-Sensoren und eines entsprechenden Netzwerkes im Gebäude. Das Ziel dieser Studie ist das Aufzeigen einer Alternative der bisherigen Methoden. Dazu werden ebenfalls Probe-Requests von Smartphones, aber auch Computern und Laptops, aufgezeichnet. Zum Aufzeichnen dieses Datenverkehrs verwenden *Vattapparamban et al.* das Wi-Fi Pineapple. Dazu wurde das Gebäude in Raster unterteilt, in welchem sich jeweils ein Wi-Fi Pineapple befand. Nebem dem Wi-Fi Pineapple als Hardware wurde die Software tcpdump genutzt, um das Datenaufkommen aufzuzeichnen. Die gesammelten Daten wurde zuerst auf einer SD-Karte gesichert und am Ende des Tages einem Linux-Server übergeben, um diese zentral verarbeiten zu können. Alle Pakete wurden vorher nach Probe-Requests gefiltert.
Es wurde festgestellt, dass durch das einfache Zählen der Probe-Requests je Wi-Fi Pineapple es möglich ist, die Anwesenheit von Personen zu ermitteln. Dabei ist jedoch auf den Umstand zu achten, dass auch andere Geräte wie bspw. Drucker mit WLAN-Funktion ebenfalls solche Pakete versenden und das Ergebnis verfälschen können. Daher ist es wichtig entsprechende MAC-Filter zu nutzen. MAC address randomization ist ebenfalls ein wichtiger Faktor. Dies ist eine Methode vieler Smartphones, um die Privatsphäre zu sichern und das Ausnutzen von Sicherheitslücken zu erschweren. Die Adressbereiche werden dabei zufällig zugewiesen, indem das Gerät in den sleep mode geht, ohne mit einem Access Point verbunden zu sein. In der Studie von *Vattapparamban et al.* handelt es sich dabei um ein Apple iPhone mit iOS 8 Betriebssystem. In einem Versuch mit Wireshark wurde festgestellt, dass das iPhone zuerst seine „echte“ MAC-Adresse übermittelt und anschließend in den sleep mode geht um diese zu wechseln. Der Wechsel der MAC-Adresse erfolgt nicht, sobald das iPhone mit einem Access Point verbunden war. Somit muss für dieses Verfahren ein entsprechender Algorithmus erstellt werden, welcher die „originale“ MAC-Adresse filtert, um das Ergebnis nicht zu verfäl-

schen. Ein Indikator für eine neu erstellte MAC-Adresse ist die konstante Signalstärke (diese bleibt dennoch immer annähernd gleich). Des Weiteren muss der Organizationally Unique Identifier (OUI) (die ersten 3 Oktette der MAC-Adresse) mit der IEEE Standards Association übereinstimmen. Dies bestimmt den Hersteller des Geräts, dem auch dieser MAC-Adressen-Block gehört. Wireshark filtert automatisch MAC-Adressen, die diesem Kriterium nicht entsprechen [Edw16].

Vattapparamban et al. nutzten zur Realisierung dieses Projekts das Wi-Fi Pineapple welches, je nach Ausführung, von $99.99 (Nano) bis hin zu $399.99 (Tetra Tactical) kostet [Hak17]. Das Wi-Fi Pineapple ist ein Access Point, welcher neben dem Promiscuous- und Monitor-Mode u.a. auch über Funktionen wie Man-in-the-Middle-Angriffen und Keylogging verfügt [Ori13, S.206].
Dieses Vorhaben wäre ebenfalls mit dem embedded Device aus dieser Arbeit realisierbar und damit deutlich kostengünstiger. So könnte der ESP32 in den Promiscuous Mode versetzt werden und die MAC-Adressen aus den Probe-Response filtern. Diese werden in einer Datei abgelegt. Da es sich hier nicht um ein großes Datenaufkommen handelt, welches komplett abgesichert werden muss, kann diese Datei auch im Flashspeicher des ESP32 abgelegt werden. Somit kann vollständig auf eine SD-Karte verzichtet werden. Bei jedem neuen Paket wird die MAC-Adresse mit Adressen in der Datei abgeglichen, um Duplikate zu erkennen und den Datensatz ohne Redundanz zu aktualisieren. Zum Identifizieren von MAC address randomization müssten entsprechende Filter-Algorithmen verwendet werden. Eine Hersteller-Liste mit entsprechenden MAC-Adressen ist bspw. auf www.wireshark.org [13] zu finden. Zur Datenablage kann eine MySQL-Datenbank genutzt werden, welche über HTTP GET requests vom ESP32 angesprochen werden kann. Hierzu ist es jedoch notwendig, dass der ESP32 über eine aktive Internet-Verbindung verfügt. Ebenfalls ist zu beachten, dass der ESP32 die Internetverbindung abbricht, sobald dieser sich gleichzeitig im Promiscuous Mode befindet und als Client fungiert, um eine Internetverbindung herzustellen.

Eine weitere Studie von *Turk et al.* (2016) beschreibt einen Wireless Packet Monitor, welcher auf Basis eines Raspberry Pis konzipiert wurde.
Der Raspberry Pi zeichnet den ankommenden Datenverkehr mit einer USB-Netzwerkkarte auf und sendet diesen zum Parsen an einen Server, auf welchem eine MySQL-Datenbank aufgesetzt ist. Ein Monitor-Mode-fähiger WLAN USB-Adapter ist notwendig, da der Raspberry Pi selber nicht in den Monitor Mode versetzt werden kann. Die Übergabe des Kanals wird in einem Skript abgehandelt und erfolgt entweder zufällig oder als statischer Wert. Da der Raspberry Pi über mehrere USB-Ports verfügt, können theoretisch auch mehrere USB-Adapter in Betrieb genommen werden. Als Alternative schlagen *Turk et al.* die Verwendung eines USB-Hubs mit Netzteil vor. Auf diesem Weg wäre es möglich, jedem WLAN USB-Adapter einen Kanal zuzuweisen. Eine libp-

[13] https://www.wireshark.org/tools/oui-lookup.html, letzter Zugriff: 30.08.2017
https://code.wireshark.org/review/gitweb?p=wireshark.git;a=blob_plain;f=manuf, letzter Zugriff: 30.08.2017

cap Applikation, welche in der Programmiersprache C geschrieben wurde, zeichnet über die Adapter den Netzwerkverkehr auf. Der Server überwacht die jeweiligen Ports (mittel Netcat realisiert) und speichert die .pcap-Daten. Um die Daten in Echtzeit zu parsen, wird das in Python geschriebene Programm Pcap parser[14] genutzt. Dieser Parser schreibt die Statistik (prozentualer Anteil von Frame-Typen und -Subtypen sowie die Gesamtanzahl aller Pakete eines Access Points) mittels MySQL Queries in eine Datenbank und die SSID sowie MAC-Adresse in eine Textdatei. Laut *Turk et al.* werden die Aufgaben aufgeteilt, um die Parsing-Geschwindigkeit zu erhöhen: Da die Verarbeitung von solchen Queries je nach Datengröße eine gewisse Zeit in Anspruch nehmen, werden diese auf das Notwendige reduziert. Aus diesem Grund werden in der Datenbank keine kompletten Pakete abgelegt, sondern lediglich die wichtigsten Informationen. In mehreren Versuchen konnte *Turk et al.* feststellen, dass der Raspberry Pi 500.000 Pakete in 24,7 Minuten parsen konnte. Dadurch schlussfolgert *Turk et al.* (2016), dass 1.000.000 Pakete in einer Stunde ohne Verzögerung verarbeitet werden können, was eine Echtzeitverarbeitung ermöglicht. 80 Prozent aller Pakete hatten eine Dateigröße zwischen 20 und 320 Bytes [TDG14].

Nach der Studie von *Turk et al.* ist der Raspberry Pi im Bezug zur Aufzeichnungs-Geschwindigkeit als Sniffing-Gerät besser geeignet als der ESP32. Aus den 500.000 Paketen in 24,7 Minuten lässt sich die Geschwindigkeit in Mbps und kB/s ableiten:

$$\frac{\left(\frac{170\ \text{Byte} \cdot 500.000 \cdot 8}{1482\ \text{Sekunden}}\right)}{1.000.000} = \frac{458839\ \text{bps}}{1.000.000} = \underline{\underline{0,4588\ \text{Mbps}}}$$

$$\frac{0,4588\ \text{Mbps}}{8} = 0,0573\ \text{MB/s} \cdot 1024 = \underline{\underline{58,7314\ \text{kB/s}}}$$

In der Veröffentlichung von *Turk et al.* wird nicht auf die Gesamtgröße der 500.000 Pakete eingegangen, daher wird aus der Information „While analyzing the capture files, we have observed that [...] 80% of the packets had size varied between 20 and 320 Bytes.“ [TDG14] entnommen, dass die durchschnittliche Paketgröße bei 170 Bytes liegt. Ebenfalls erwähnen *Turk et al.* nicht, wieviele WLAN USB-Adapter genutzt wurden und wie die Modellbezeichnung dieser ist. Optional zur Datenbank könnten die Daten auch auf einem externen Speicher abgelegt werden. Im Gegensatz zum ESP32 besitzt der Raspberry PI USB-Ports, somit sind externe Festplatten zur Datenablage denkbar (Preis/Speicher-Verhältnis ist hier besser als bei SD-Karten).
Die Verwendung eines Raspberry Pis ist bezuüglich zur Geschwindigkeit eine Alternative zum ESP32, da der Raspberry Pi um den Faktor 86 schneller ist als der ESP32. Jedoch ist hier klar das Einsatzgebiet vom embedded Device zu hinterfragen. Falls le-

[14] https://pypi.python.org/pypi/pcap-parser, letzter Zugriff: 30.08.2017

diglich eine geringe bzw. bestimmte Anzahl an Paketen verarbeitet werden müssen, ist Punkt Kosten der ESP32 dem Versuchsaufbau von *Turk et al.* vorzuziehen.

Des Weiteren wird in der Studie von *Acuna et al.* (2017) Möglichkeiten eines Sniffing-Device in Kombination mit Drohnen vorgestellt. Ausgangspunkt der Studie ist die Frage, ob es möglich sei, Probe-Requests von Smartphones u.a. mithilfe von Drohnen zu empfangen, um diese anschließend zu orten. Dabei soll eine Drohne mittels maschinellen Lernens und Trainingsdaten in einem definierten Bereich Probe-Requests Pakete empfangen, um anschließend die Geräte aufzuspüren. Dieser Bereich wird in sechs Zonen unterteilt. Das Ziel von *Acuna et al.* war es, den Smartphones die korrekte Zone zuzuordnen, in welcher diese sich befinden.
Bei den notwendigen Bestandteilen eines Probe-Requests, welche zum Orten eines Gerätes notwendig sind, handelt es sich um die Signalstärke und die MAC-Adresse. Wie auch in der Veröffentlichung von *Vattapparamban et al.* (2016), wird auch im Vorhaben von *Acuna et al.* das Wi-Fi Pineapple als Aufzeichnungsgerät genutzt. Dieses wird an einer „Tarot 650“ Drohne befestigt und über Batterien mit Energie versorgt. Um das Wi-Fi Pineapple aus der Entfernung über eine VPN-Verbindung zu bedienen, wurde von Acuna et. al ein Smartphone an der Drohne befestigt, welche das Wi-Fi Pineapple mit einer Internetverbindung versorgt. Über die damit mögliche VPN-Verbindung und ein Web-Interface wird das Programm tcpdump aus der Ferne ausgeführt. Sobald das Wi-Fi Pineapple eine Internetverbindung erfolgreich aufbauen konnte, wird Datum und Uhrzeit über einen NTP-Server geupdatet. Sofern die Initialisierung erfolgreich abgeschlossen ist, startet die Paketaufzeichnung, welche den Zeitstempel, die MAC-Adresse (+ Herstellerbezeichnung) und die Signalstärke beinhaltet. Mittels der VPN-Verbindung werden die Daten an einen Server übertragen. Die Zeit vom Wi-Fi Pineapple und der Drohne werden synchronisiert, so entspricht der Zeitstempel der empfangenen Pakete den GPS-Koordinaten. Die Drohne kommuniziert die Telemetrie-Daten an die Software „Mission Planer“, sodass die aktuelle Position jederzeit überprüft werden kann. Um der Drohne autonomes Fliegen zu ermöglichen, wird diese mit einem „Pixhawk Flight Controller“ ausgestattet, welcher durch Bildverarbeitung eine Autopilot-Erweiterung bietet. Hierzu wurde die Dohne mit dem Random Forest Algorithmus antrainiert, welches ein Klassifikationsverfahren im Bereich des maschinellen Lernens darstellt. Vorteile bei diesem Klassifikator ist die sehr schnelle Traningszeit, hohe Genauigkeit und geringe Fehlerrate. Mittels dieses Verfahrens konnten *Acuna et al.* eine Lokalisierungs-Genauigkeit von 81,8 Prozent erzielen. Die Genauigkeit (Accuracy) beschreibt, wie viele Probe Requests in der Menge korrekt der richtigen Zone zugeordnet wurden und berechnet sich mittels:

$$Accuracy = \frac{N_{TP} + N_{TN}}{N_{Total}}$$

wobei N_{TP} die Anzahl der true positives (TP) Probe-Requests ist, also solche Probe-Requests, welche richtig dem aktuellen Raster zugeordnet wurden. N_{TN} ist die Anzahl der true negatives (TN) Probe-Requests. Dies sind alle Pakete, welche richtig einem

anderen Raster als dem aktuellen der Drohne zuzuordnen sind. Die Gesamtanzahl aller Probe-Requests wird mit N_{Total} beschrieben [Acu17].

In der Studie von *Acuna et al.* wurde zur Detektierung von Probe-Requests das Wi-Fi Pineapple genutzt. Die Daten wurden anschließend über eine VPN-Verbindung an einen Server übertragen. Es wäre vorstellbar in diesem Einsatzbereich das Wi-Fi Pineapple durch das embedded Device aus dieser Arbeit zu ersetzen. Das embedded Device ist ebenfalls in der Lage Probe-Requests aufzuzeichnen und nach den wichtigsten Eigenschaften zu filtern. Dabei sind folgende Faktoren jedoch zu beachten:

Reichweite
Die Reichweite des ESP32 wurde in dieser Bachelorarbeit nicht weiter analysiert. Auch im Datenblatt des ESP32 sind keine weiteren Informationen zu finden, wie der Empfang von Datenpaketen sich in Relation zur Entfernung verhält. Theoretisch ist in der Arbeit von *Acuna et al.* das Wi-Fi Pineapple durch einen ESP32 ersetzbar, jedoch bedarf dies weiterer Forschung seitens der Signalqualität des Wi-Fi Chips. Williams beschreibt in einem Artikel auf der Internetplattfom Hackaday, wie der Nutzer „Jeija“ die Reichweite des ESP32 mithilfe einer Richtantenne auf bis zu 10km erhöhen konnte [Wil17].

Zeitstempel
Die RTC des ESP32 kann auf bspw. mit dem Network Time Protocol (NTP) auf ein Datum und eine Uhrzeit eingestellt werden. Dazu muss sich der ESP32 als Client mit einem Access Point kurzzeitig verbinden, um diese Informationen beziehen zu können. Sobald dies geschieht und gleichzeitig der Promiscuous Mode des ESP32 aktiviert ist, bricht die Wi-Fi Verbindung des ESP32 ab. Das Beziehen der aktuellen Zeit und das Aktivieren des Promiscuous Modes muss daher nacheinander geschehen. Wie auch das Wi-Fi Pineapple benötigt der ESP32 hierfür eine aktive Internetverbindung, welche ebenfalls mit einem Smartphone realisiert werden kann. Alternativ kann ein GSM Shield an den ESP32 angebracht werden, sodass dieser das Mobilfunknetz dafür nutzen kann. [15]

Datenübertragung
Eine VPN-Verbindung ist theoretisch mit dem ESP32 möglich, jedoch bricht hier ebenfalls bei einer gemeinsamen Nutzung des Promiscuous Mode die Verbindung ab. Dies wurde bereits in der Studie von *Vattapparamban et al.* (2016) angesprochen. Somit müsste der Promiscuous Mode erst deaktiviert werden, anschließend die Daten über eine VPN-Verbindung übertragen werden und dann der Promiscuous Mode wieder aktiviert werden.

Die Arbeit von *Acuna et al.* wäre ebenfalls mit dem ESP32 realisierbar. Dies würde jedoch die Modifikation des embedded Device erfordern. Je nach Flughöhe der Drohne wäre eine Richtantenne notwendig, um den Empfang von Paketen zu verbessern. Damit

[15] https://store.arduino.cc/arduino-gsm-shield-2-integrated-antenna, letzter Zugriff: 02.09.2017
https://www.arduino.cc/en/Guide/ArduinoGSMShield#toc10, letzter Zugriff: 02.09.2017

würde auch bzgl. der Drohne der Faktor Gewicht hinzukommen. Die Berechnung des Random Forest Algorithmus übernimmt der Server, nicht das Wi-Fi Pineapple. Dies bedeutet, dass die Berechnung des Klassifikators ebenfalls nicht vom ESP32 übernommen werden müsste. Die Verwendung eines Wi-Fi Pineapples ist kostenintensiv. Darüber hinaus würde die Verwendung einer solchen Technologie den Rahmen, im Verhältnis zur Zielerreichung dieser Studie, überproportional erhöhen.

Abschließend lässt sich sagen, dass insbesondere sind die Ansätze des CACFA- bzw. CAOCA-Verfahren von *Qin et al.* (2013) als Implementierung in der vorliegenden Arbeit hervorragend geeignet. Das beschriebene System würde auch ohne Kenntnis des genutzten Kanals sicherstellen, dass dieser vom embedded Device autonom festgestellt werden kann. Es gewährleistet im Gegensatz zum automatischen Kanalwechsel zudem weniger Datenverlust. Im Gegensatz zum automatischen Kanalwechsel gewährleistet dies weniger Datenverlust. Von *Vattapparamban et al.* und *Acuna et al.* werden mögliche Einsatzgebiete im Bereich Smart-Building und Drohnen vorgestellt, welche im Abschnitt 5.4 weiter erläutert werden. *Turk et al.* [TDG14] konnte im Vergleich zum embedded Device dieser Arbeit wesentlich mehr Daten aufzeichnen. Der deutliche Vorteil des Raspberry Pis liegt in der stetigen Modifizierbarkeit. So können stets weitere WLAN USB-Adapter über USB-Hubs an den Einplatinencomputer angeschlossen werden, um das Benchmark-Ergebnis zu verbessern.

5.2 Zusammenfassung

Mit dieser Arbeit wurden die notwendigen elementaren Grundlagen und Konzepte erläutert, welche zur Konzeptierung eines embedded Device zur Aufzeichnung von Datenverkehr in Drahtlosnetzwerken notwendig sind.
Ebenfalls wurden mögliche Schaltpläne zur Realisierung eines solchen Projekts aufgezeigt. Kernstück dieses Device stellt der ESP32 dar, welcher mittels des Espressif IoT Development Frameworks geflasht wird.

Bei der Verwendung des ESP-IDFs ist es notwendig, die aktuelle Version (2.1, Stand August 2017) zu verwenden, da im commit[10] `b260f6cb25bf7ab9be907fe4fdebaac7aacb8900` vom 07.08.2017 ein für dieses Projekt essenzieller Fehler ausgebessert wurde. So war es vor dieser Änderung nicht möglich, einen Buffer fehlerfrei im 1-line oder 4-line Mode auf eine SD-Karte zu speichern. Beim Versuch diese Daten zu speichern kam es zu Formatierungsfehlern wie fehlende Leerzeichen und Absätze, sowie doppelt oder gar nicht auftretende Zeichen.

Eine alternative Möglichkeit zum Flashen des Moduls bietet sich in Form der Arduino IDE an. Aufgrund der Verwendung des „Arduino cores“ als Komponente, kann der vollständige Programmcode (einschließlich den Headerdateien `Deauth` und `HexToBinary`)

[10] Bezeichnet das Speichern des gegenwärtigen Status im Projekt

ohne zusätzliche Modifikation innerhalb dieser Umgebung genutzt werden. Dabei kommt es jedoch zu Einschränkungen bezüglich hardwarenahen Einstellungen: Die Frequenz der CPU des ESP32 kann nicht angepasst werden, Konfiguration bezüglich Watchdog, Brownout Detection, FAT Filesystem, Wi-Fi und FreeRTOS sind nicht möglich. Da somit die Zeichensatztabelle und die maximale Zeichenlänge nicht definierbar sind, stehen default maximal 8 Zeichen (inkl. Dateiendung) für den Dateinamen der Capture-File zur Verfügung. Auch kann FreeRTOS nicht konfiguriert werden, selbst wenn dies auf einem oder beiden Kernen läuft.

Bei der Verwendung einer SD-Karte ist auf den teilweise (je nach Hersteller) verbauten CP2102 Mikrochip hinzuweisen, da dieser inkompatibel zum Prozessor ist. Dieser ist beispielsweise mit einem FT2232 oder USB3300 Mikrochip auszutauschen, um die korrekte Verwendung des 4-line Modes zu gewährleisten. Bezugnehmend auf die Geschwindigkeit des embedded Device (0,005284 Mbps bzw. 0,676 kB/s) ist dies jedoch vernachlässigbar, da das arithmetische Mittel des Benchmarks der SD-Karte eine Schreibgeschwindigkeit von 1,2 MB/s ergab. Die Schreibgeschwindigkeit variiert außerdem durch verwendete SD-Karte, SD-Karten Adapter und Leitung (sowohl Länge als auch Qualität) zwischen Adapter und ESP32.

Um möglichst viele Pakete eines spezifischen Access Points aufzuzeichnen, ist es notwendig auf dem richtigen Kanal zu sniffen, dadurch kann auf den automatischen Kanalwechsel verzichtet werden. Je nachdem, wie groß der zeitliche Abstand zwischen dem Kanalwechsel ist, werden Pakete nicht aufgezeichnet. Sollte beispielsweise der Kanalwechsel eingestellt sein und zwischen jedem Wechsel 10ms liegen, werden in einem Zeitraum von $10ms \cdot (13-1) = \underline{\underline{120ms}}$ keine Pakete aufgezeichnet. In jedem Fall entspricht dies unabhängig von der Dauer bis zu über 90% Paketverlust. Eine Variante den genutzten Kanal zu überprüfen liegt in Wireshark.

So könnte der Netzwerkverkehr mit automatischen Kanalwechsel aufgezeichnet und die erstellte Capture-File in Wireshark kontrolliert werden. Anschließend wird der entsprechende Kanal im Programmcode implementiert und ausschließlich dieser überwacht. Hier ist jedoch zu erwähnen, dass dieses System nur dann funktioniert, wenn der Access Point nicht automatisch den Kanal wechselt (z.B. bei einer schlechten Verbindung auf dem aktuellen Kanal).

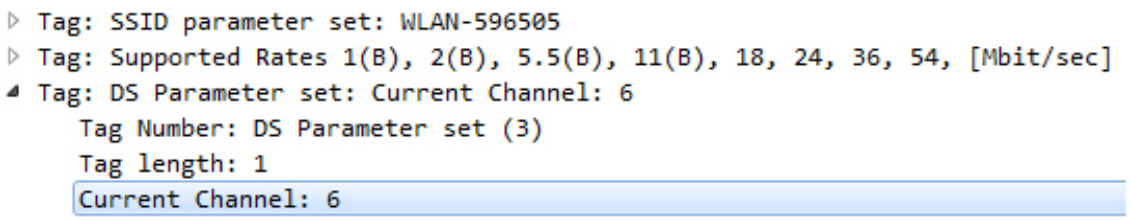


Abbildung 5.1: Überprüfen des Kanals in Wireshark

Eine weitere Möglichkeit ist die Umsetzung der von *Qin et al.* (2013) vorgestellten Verfahren zur autonomen Kanalbestimmung. Weitere mögliche Fehlerquellen wären:

Brownout
Während des Flashens, aber auch im Betrieb, kann ein Brownout auftreten. Sobald die Versorgungsspannung des MCU unterschritten wird, fallen nach und nach Komponenten aus. Dies wird über die Fehlermeldung

```
Brownout detector was triggered
```

im Terminal signalisiert. Ein Brownout tritt so z.B. durch leer werdende Batterien bzw. Powerbanks (und anderen Akkumulatoren) auf [Sch17b]. Auch schlecht verarbeitete USB-Kabel können diese Fehlermeldung auslösen, da bei diesem Leiter zuviel Spannung abfallen kann. Der Brownout detector kann im ESP-IDF mittels `make menuconfig` unter:

Component config —> ESP32-specific —> Hardware brownout detect & reset

deaktiviert werden. Bei Bedarf kann auch die Spannung als Parameter übergeben werden, bei welcher der Brownout detector eine Fehlermeldung ausgibt.

Watchdog
Der Watchdog überwacht den korrekten Funktionsablauf eines Mikrocontrollers. Sobald dieser vom Prozessorkern für einen definierten Zeitraum kein Signal erhält, geht dieser von einer Fehlfunktion aus.
Der Watchdog verfügt über einen Zähler mit einem Startwert und dekrementiert diesen. In regelmäßigen Abständen wird auf eine Adresse im Adressraum des Prozessorkerns zugegriffen. Ist dies erfolgreich, wird der Zähler wieder auf den Startwert zurückgesetzt. Sofern der Zähler „0“ erreicht, wird dies als ein Fehler im Programmablauf gewertet und der Prozessorkern in einen Grundzustand zurück gesetzt, sowie der Programmablauf neu gestartet [Ung13, S. 157]. Der Watchdog kann über `make menuconfig` deaktiviert werden:

Component config —> ESP32-specific —> Task Watchdog
Component config —> ESP32-specific —> Interrupt Watchdog

5.3 Fazit

Zusammenfassend kann gesagt werden, dass das Ziel ein embedded Device zu entwickeln, welches in der Lage ist Netzwerkverkehr in Drahtlosnetzwerken aufzuzeichnen, im Zuge dieser Bachelorarbeit erfolgreich umgesetzt werden konnte. Zu diesem Zweck wurde der ESP32 genutzt, um ein entsprechendes System aufzubauen. Dabei ergab sich, dass die Aufzeichnungsgeschwindigkeit laut Benchmark bei 0,005284 Mbps liegt, was

0,676 kB/s entspricht. Einerseits ist dies auf die CPU-Taktrate des ESP32 zurück zu führen, welche bei maximal 240 MHz liegt, als auch andererseits auf die Verarbeitung der Daten im Ringpuffer. Nach dem mooreschen Gesetz[16] sollte Ersteres in absehbarer Zeit der Vergangenheit angehören. Was die Datenverarbeitung betrifft, so könnte ein effizienter Puffer notwendig sein, um den Paketverlust so gering wie nur möglich zu halten. Auch eine wirkungsvollere Aufteilung der einzelnen Schritte des Programmablaufs auf die beiden Kerne könnte hier zielführend sein. Das vollständige Aufzeichnen des Netzwerkverkehrs ist, je nach Datenaufkommen, mit dem embedded Device daher zwar nicht möglich, jedoch gibt es zahlreiche andere Einsatzgebiete, auf welche im Ausblick 5.4 genauer eingegangen werden.

5.4 Ausblick

Das embedded Device ist stets modifizierbar und auf den jeweiligen Einsatz anpassbar. Eine mögliche Modifizierung ist die Parameterübergabe: Die aktuelle Übergabe der Parameter erfolgt direkt innerhalb des Programmcodes. Alternativ kann eine `Kconfig.projbuild` Datei im main Ordner des Projekts abgelegt werden. Dadurch ist es möglich unter `make menuconfig` eigene Menüpunkte (und Unterpunkte) zu erstellen um dort die vollständige Parameterübergabe abzulegen. Eine Modifikation der Datei `esp32_promiscuous.cpp` wäre somit nicht mehr notwendig.

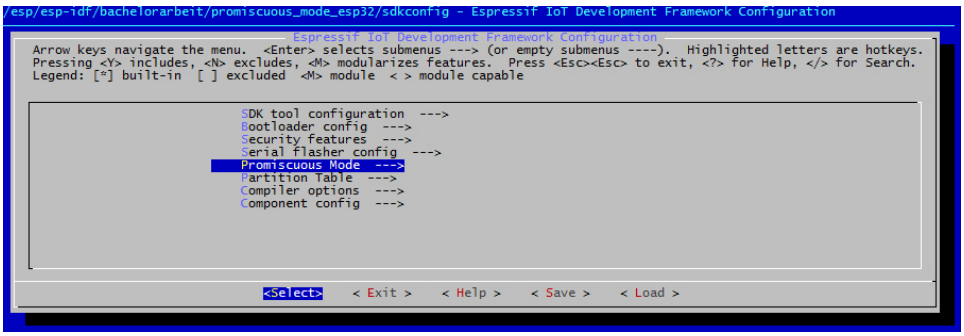


Abbildung 5.2: Promiscuous Mode als Menüpunkt zur Parameterübergabe

Im Anhang B.4 ist ein Beispiel einer solchen Datei zu finden.

Eine weitere Variante ist die Übergabe der notwendigen Parameter über eine Web-Oberfläche. So könnte der ESP32 als Access Point verwendet werden. Nach einem erfolgreichen Login auf diesen Access Point, ist die Web-Oberfläche über die IP-Adresse des ESP32 erreichbar, worüber alle Parameter übergeben werden. Der MAC-Filter wird darüber hinaus durch eine Scan-Funktion ersetzt. Alle Access Points in der unmittelbaren Umgebung werden gescannt und anschließend aufgelistet. So kann der Access Point,

[16] „Die Komplexität integrierter Schaltkreise verdoppelt sich alle zwölf bis 24 Monate“ [Sum16, S. 4]

welcher gesnifft werden soll, direkt ausgewählt werden, ohne dass die MAC-Adresse vorher bekannt ist. Die Konfiguration kann somit über einen Computer/Laptop, aber auch über ein Smartphone erfolgen, sodass der Aspekt der Mobilität erhalten bleibt.
Ein weiterer wichtiger Aspekt ist die Erhöhung der Sniffing-Geschwindigkeit, sodass weitaus mehr Netzwerkverkehr aufgezeichnet werden kann. Derzeit ist dies zum einen durch die Taktrate der CPU beschränkt, als auch durch die Verarbeitung des Puffers.

Vattapparamban et al. und *Qin et al.* erläutern in Ihren Studien Einsatzfelder von Wireless Sniffern in sogenannten Smart Buildings. Hier steht vor allem das Thema der effizienten Energienutzung im Vordergrund. Ein weiterer Bereich wäre die Sicherheit von Personen in diesem Gebäude. Bei einem Brand oder ähnlichen Katastophen wäre es den Einsatzkräften möglich, sich in Echtzeit ein Bild der aktuellen Gefahrensituation zu machen und zu Beurteilen in welchen Räumen sich noch Personen befinden. Selbstverständlich setzt dies bei den besagten Personen ein Gerät vorraus, welches entsprechende Datenpakete autark versendet (bspw. Smartphones), daher stellt dieses System keine 100%ge Sicherheit dar.
Des Weiteren ist es vorstellbar, unbemannte Luftfahrzeuge (beispielsweise Drohnen) mit dem embedded Device auszustatten. Eine solche Variante könnte Probe-Request Pakete von Smartphones aufzeichnen und über die Signalstärke. Denkbare Einsatzgebiete könnten u.a. Vermisstensuche und Fahndungen sein.

Literaturverzeichnis

[Acu17] Acuna et al.: Localization of WiFi Devices Using Probe Requests Captured at Unmanned Aerial Vehicles. In: *IEEE Xplore* (2017). http://dx.doi.org/10.1109/WCNC.2017.7925654. – DOI 10.1109/WCNC.2017.7925654

[Aut12] Autorenteam, C.: *Informatik für technische Kaufleute und HWD.* Compendio Bildungsmedien, 2012 (Compendio Bildungsmedien. Informatik). – ISBN 9783715596525

[Bad15] Badach, Erwin Anatol und H. Anatol und Hoffmann: *Technik der IP-Netze.* Carl Hanser Verlag GmbH & Co. KG, 2015. – ISBN 9783446439863

[Bal12] Ballmann, B.: *Network Hacks - Intensivkurs: Angriff und Verteidigung mit Python.* Springer Berlin Heidelberg, 2012 (Xpert.press). – ISBN 9783642243059

[Bau15] Baun, Christian: *Computernetze kompakt.* Springer Berlin Heidelberg, 2015. – ISBN 9783662469323

[Bol17] Bollhöfer, E.: *Schutz von Unternehmensdaten bei der Erbringung von E-Services: Rechtliche, technische und organisatorische Lösungsansätze für Unternehmen des Maschinen- und Anlagenbaus.* Springer Fachmedien Wiesbaden, 2017 (Research (Wiesbaden, Germany)). – ISBN 9783658184865

[BS03] Bellardo, John ; Savage, Stefan: 802.11 Denial-of-service Attacks: Real Vulnerabilities and Practical Solutions. In: *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12.* Berkeley, CA, USA : USENIX Association, 2003 (SSYM'03), 2–2

[Bä10] Bähring, Helmut: *Anwendungsorientierte Mikroprozessoren. Mikrocontroller und Digitale Signalprozessoren.* Bd. 4. Springer-Verlag Berlin Heidelberg, 2010. – ISBN 9783642122927

[Dem15] Dembowski, Klaus: *Raspberry Pi - Das technische Handbuch.* Springer Fachmedien Wiesbaden, 2015. – ISBN 9783658087111

[Edw16] Edwin Vattapparamban et al.: Indoor Occupancy Tracking in Smart Buildings Using Passive Sniffing of Probe Requests. In: *IEEE Xplore* (2016). http://dx.doi.org/10.1109/IWCMC.2016.7577060. – DOI 10.1109/IWCMC.2016.7577060

[Esp17a] Espressif Systems: *Arduino core for ESP32 Wi-*

Fi chip. 2017. – https://github.com/espressif/arduino-esp32/tree/87093368d9b42d8a211d95ff54b8d3d2e26120d2#using-as-esp-idf-component,letzterZugriff:17.05.2017

[Esp17b] Espressif Systems: *ESP-IDF Programming Guide*. 2017. – https://esp-idf.readthedocs.io/en/latest/index.html,letzterZugriff:17.05.2017

[Esp17c] Espressif Systems: *ESP-WROOM-32 Datasheet*. 2017. – https://www.espressif.com/sites/default/files/documentation/esp_wroom_32_datasheet_en.pdf,letzterZugriff:18.05.2017

[Esp17d] Espressif Systems: *ESP32 Datasheet*. 2017. – https://espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf,letzterZugriff:15.05.2017

[Esp17e] Espressif Systems: *ESP8266EX*. 2017. – http://espressif.com/sites/default/files/documentation/0a-esp8266ex_datasheet_en.pdf,letzterZugriff:30.08.2017

[Esp17f] Espressif Systems: *SDMMC Host Peripheral*. 2017. – https://esp-idf.readthedocs.io/en/v2.0/api/storage/sdmmc.html#c.SDMMC_FREQ_HIGHSPEED, letzter Zugriff: 15.08.2017

[Fin11] Finneran, M.F.: *Communications engineering series*. Bd. 1: *Voice Over WLANS: The Complete Guide*. Elsevier Science, 2011. – ISBN 9780080556437

[Fra08] Franke, Andreas: *WLAN Einsatz an der HTWK-Leipzig*. Diplom.de, 2008 http://www.ebook.de/de/product/21622803/andreas_franke_wlan_einsatz_an_der_htwk_leipzig.html. – ISBN 9783836612524

[Gra17] Gratton, Angus: *SD Card example*. 2017. – https://github.com/espressif/esp-idf/blob/master/examples/storage/sd_card/README.md, letzter Zugriff: 31.05.2017

[GWUW17] Gehrke, Winfried ; Winzker, Marco ; Urbanski, Klaus ; Woitowitz, Roland: *Digitaltechnik. Grundlagen, VHDL, FPGAs, Mikrocontroller*. Springer-Verlag GmbH, 2017. – ISBN 978–3–662–49730–2

[Hak17] HakShop by Hak5: *WiFi Pineapple*. 2017. – https://hakshop.com/products/wifi-pineapple,letzterZugriff:26.08.2017

[Har13] Hartmann, Detlef: *Geschäftsprozesse mit Mobile Computing: Konkrete Projekterfahrung, technische Umsetzung, kalkulierbarer Erfolg des Mobile Business*. Springer-Verlag, 2013. – ISBN 9783322902757

[HM16] Hurer-Mackay, William: *Forging WiFi Beacon Frames Using Scapy*. 2016. – https://www.4armed.com/blog/forging-wifi-beacon-frames-using-scapy/, letzter Zugriff: 27.08.2017

[Hof05] Hofherr, M.: *WLAN-Sicherheit: Professionelle Absicherung von 802.11-Netzen*. Bd. 2. Heise, 2005. – ISBN 9783936931259

[Hus10] Husar, P.: *Biosignalverarbeitung*. Springer Berlin Heidelberg, 2010. – ISBN 9783642126574

[Kam12] Kammermann, Markus: *CompTIA Network+*. mitp-Verlag, 2012. – ISBN 9783826692048

[Kau08] Kauffels, Franz-Joachim: *Lokale Netze*. Redline Wirtschaft, 2008. – ISBN 9783826659614

[Kry15] Krypzcyk, V.: *PIC-Mikrocontroller: Grundlagen und Praxisworkshop*. entwickler.Press, 2015 (shortcuts). – ISBN 9783868025385

[LAN17a] LANCOM Systems: *Short Guard Interval*. 2017. – https://www.lancom-systems.de/docs/LCOS-Refmanual/9.10-Rel/DE/topics/aa1399779.html, letzter Zugriff: 22.05.2017

[Lan17b] Langmann, Reinhard: *Taschenbuch der Automatisierung*. Carl Hanser Verlag GmbH & Co. KG, 2017. – ISBN 9783446451025

[LC08] Lee, Byeong G. ; Choi, Sunghyun: *Broadband Wireless Access & Local Networks*. Artech House, 2008. – ISBN 9781596932944

[MS11] Meinel, Christoph ; Sack, Harald: *Internetworking*. Springer Berlin Heidelberg, 2011. – ISBN 9783540929406

[Nor17] Nordic Semiconductor ASA: *QSPI — Quad serial peripheral interface*. 2017. – http://infocenter.nordicsemi.com/index.jsp?topic=%2Fcom.nordic.infocenter.nrf52840.ps%2Fqspi.html,letzterZugriff:22.05.2017

[Ori13] Oriyano, S.P.: *Hacker Techniques, Tools, and Incident Handling*. Jones & Bartlett Learning, LLC, 2013 (Jones & Bartlett Learning information systems security & assurance series). – ISBN 9781284031706

[Pre11] Prevezanos, Christoph: *Computer Lexikon 2012*. Markt+Technik Verlag, 2011. – ISBN 3827247284

[QZLS13] Qin, Weijun ; Zhang, Jiadi ; Li, Bo ; Sun, Limin: Discovering Human

Presence Activities with Smartphones Using Nonintrusive Wi-Fi Sniffer Sensors: The Big Data Prospective. In: *International Journal of Distributed Sensor Networks* (2013). http://dx.doi.org/10.1155/2013/927940. – DOI 10.1155/2013/927940

[Sal16] Salazar, Jordi: *Drahtlose Netzwerke*. 2016. – techpedia.fel.cvut.cz/download/?fileId=435&objectId=75, letzter Zugriff: 10.05.2017

[Sch17a] Schwarz, Andreas: *AVR-Tutorial: Timer*. 2017. – https://www.mikrocontroller.net/articles/AVR-Tutorial:_Timer, letzter Zugriff: 01.06.2017

[Sch17b] Schwarz, Andreas: *Brownout*. 2017. – https://www.mikrocontroller.net/articles/Brownout, letzter Zugriff: 15.08.2017

[Sch17c] Schwarz, Andreas: *Oszillator*. 2017. – https://www.mikrocontroller.net/articles/Oszillator, letzter Zugriff: 05.06.2017

[Som12] Sommer, Ulli: *Arduino. Mikrocontroller-Programmierung mit Arduino/Freeduino*. Franzis Verlag, 2012. – ISBN 9783645250351

[SPC08] SPC Multicomp Electronics Distributor: *BC547B General Purpose Transistor Datasheet*. 2008. – https://www.farnell.com/datasheets/410427.pdf, letzter Zugriff: 16.08.2017

[Sum16] Summa, Leila: *Digitale Führungsintelligenz: Adapt to win: Wie Führungskräfte sich und ihr Unternehmen fit für die digitale Zukunft machen*. Springer Fachmedien Wiesbaden, 2016. – ISBN 9783658108021

[SZ14] Siegl, Johann ; Zocher, Edgar: *Schaltungstechnik - Analog und gemischt analog/digital*. Springer Vieweg, 2014. – ISBN 978–3–642–29559–1

[TDG14] Turk, Yusuf ; Demir, Onur ; Gören, Sezer: Real Time Wireless Packet Monitoring with Raspberry Pi Sniffer. In: *Information Sciences and Systems 2014 pp 185-192* (2014). http://dx.doi.org/10.1109/IWCMC.2016.7577060. – DOI 10.1109/IWCMC.2016.7577060

[Ung13] Ungerer, T.: *Mikrocontroller und Mikroprozessoren*. Springer Berlin Heidelberg, 2013 (Springer-Lehrbuch). – ISBN 9783662087466

[Wil17] Williams, Al: *ESP32 WiFi Hits 10km with a Little Help*. 2017. – https://hackaday.com/2017/04/11/esp32-wifi-hits-10km-with-a-little-help/, letzter Zugriff: 02.09.2017

[Wir17a] Wireshark-Community: *Broadcast*. 2017. – https://wiki.wireshark.org/Broadcast, letzter Zugriff: 03.06.2017

[Wir17b] Wireshark-Community: *Multicast*. 2017. – https://wiki.wireshark.org/Multicast, letzter Zugriff: 03.06.2017

[Wir17c] Wireshark-Community: *Unicast*. 2017. – https://wiki.wireshark.org/Unicast, letzter Zugriff: 03.06.2017

[Wir17d] Wireshark-Community: *WLAN (IEEE 802.11) capture setup*. 2017. – https://wiki.wireshark.org/CaptureSetup/WLAN, letzter Zugriff: 03.06.2017

[WR16] Wallace, S. ; Richardson, M.: *Getting Started With Raspberry Pi: An Introduction to the Fastest-Selling Computer in the World*. Maker Media, Incorporated, 2016. – ISBN 9781680452426

Anhang

A Tabellen

A.1 OSI-Referenzmodell

Tabelle A.1: OSI-Referenzmodell [modifiziert nach [Aut12, S. 84]]

Schicht	OSI-Bezeichnung	TCP/IP Stack	Beispiele
7	**Anwendungsschicht** (Application Layer)	**Application Layer** Auf dieser Ebene werden die Daten so aufbereitet, dass sie an eine anderen Anwendung im Netzwerk übertragen werden können.	HTTP, FTP, POP, Telnet
6	**Darstellungsschicht** (Presentation Layer)		
5	**Sitzungsschicht** (Session Layer)		
4	**Transportschicht** (Transport Layer)	**Transport Layer** Diese Ebene garantiert die Ablieferung und Weiterleitung der Daten	TCP, UDP
3	**Vermittlungsschicht** (Network Layer)	**Internet Layer** Weltweite einheitliche Adressierung und Weiterleitung der Daten	IP, ARP, Router
2	**Sicherungsschicht** (Data Link Layer)	**Network Access Layer** Auf dieser Ebene wird die Art der Datenübertragung bzw. Netzwerkverbindung konkretisiert	Ethernet, ISDN, MAC, Switch
1	**Bitübertragungsschicht** (Physical Layer)		

B Programmcode

B.1 esp32_promiscuous.cpp

```
/* ESP32 WLAN-Sniffer
(C) 2017, Eric Schroeder */

#include <stdio.h>
#include <string.h>
#include <inttypes.h>
#include <sys/unistd.h>
#include <sys/stat.h>
#include "freertos/FreeRTOS.h"
#include <freertos/queue.h>
extern "C" {
#include "freertos/ringbuf.h"
}
#include "esp_vfs_fat.h"
#include "driver/sdmmc_host.h"
#include "driver/sdspi_host.h"
#include "sdmmc_cmd.h"
#include "driver/gpio.h"
#include "esp_err.h"
#include "esp_log.h"
#include "esp_wifi.h"
#include "esp_wifi_types.h"
#include "esp_wifi_internal.h"
#include "esp_system.h"
#include "esp_event.h"
#include "esp_event_loop.h"
#include "nvs_flash.h"
#include "Arduino.h"
#include "HexToBinary.h"
#include "Deauth.h"

////////////////////////EINSTELLUNGEN/////////////////////////
#define FILENAME "benchmark" // Dateiname festlegen (ohne
    Dateiendung)
#define SWITCHING_TIME 50 // Zeit in ms, bis der Kanal gewechselt
    wird
#define CHANNEL_SWITCHING true // True, wenn Kanal automatisch
    gewechselt werden soll
#define CHANNEL 1 // Kanal welcher gesnifft werden soll (wenn
    CHANNEL_SWITCHING false)
#define LED_ERROR 5 // LED-Pin fuer Fehlermeldung definieren
#define LED_READY 30 // LED-Pin fuer erfolgreiche Initialisierung
    definieren
#define LED_PAKET 19 // LED-Pin fuer empfangene Pakete definieren
#define BUTTON 23 // Drucktaster-Pin fuer Start/Stop des Skripts
#define POWERBANK 21 // Pin fuer die Basis des Transistors
#define DEAUTHENTIFICATION false // True, wenn eapol 4-way
```

```
   handshake benoetigt wird (Deauth-Attacke)
uint8_t mac_filter[] = { }; // sniff specific mac adress
// Fuer Adresse 1-4, Form (Hex): { 0xff, 0xff, 0xff, 0xff, 0xff,
   0xff, }. Filter-Deaktivierung mit: { }
////////////////////////////////////////////////////////////////

uint32_t offset = 0;
uint16_t client_sniffing = 0;
uint16_t timetosafe = 0;
uint8_t client[12];
uint8_t channel = CHANNEL;
uint8_t led1 = LED_ERROR;
uint8_t led2 = LED_READY;
uint8_t led3 = LED_PAKET;
uint8_t powerbank = POWERBANK;
uint8_t button = BUTTON;
uint8_t buttonState = 0;
uint8_t Init_success = 0;
sdmmc_card_t* card;
static const char* filename = FILENAME;
static const char* TAG = "Hinweis";
char buf[256];
char hexbuf[8192];
char hexbuf2[12];
char hexbuf_dump[8192] = {};
char filenamebuf[100];
unsigned long previousMillis;
HexToBinary HexToBinary;
Deauth Deauth;


/*- type = RINGBUF_TYPE_NOSPLIT: The insertion code will leave the
   room at the end of the ringbuffer
unused and instead will put the entire item at the start of the
   ringbuffer, as soon as there is
enough free space.*/
RingbufHandle_t packetRingbuf = xRingbufferCreate(12 * 1024,
   RINGBUF_TYPE_NOSPLIT);

esp_err_t event_handler(void* ctx, system_event_t* event);
void wifi_promiscuous(void* buffer, wifi_promiscuous_pkt_type_t
   type);

void setup()
{
   sprintf(filenamebuf, "/sdcard/%s.pcap", filename);
   gpio_pullup_en(GPIO_NUM_12); //Software Pullup -> Notwendig fuer
      SD Karte (4-line)
   pinMode(led1, OUTPUT);
   pinMode(led2, OUTPUT);
   pinMode(led3, OUTPUT);
   pinMode(powerbank, OUTPUT);
```

```
88     pinMode(button, INPUT);
89 }
90
91 void save_to_sdcard(const char* data)
92 {
93     //const char* data = (const char*) param;
94     sprintf(filenamebuf, "/sdcard/%s.pcap", filename);
95     //ESP_LOGI(TAG, "Opening file");
96     FILE* f = fopen(filenamebuf, "a");
97
98     if (f == NULL) {
99         ESP_LOGE(TAG, "Fehler! Datei konnte nicht geoeffnet werden!");
100        digitalWrite(led1, HIGH);
101        if (Init_success == 0) digitalWrite(led1, LOW);
102        vTaskDelay(500 / portTICK_PERIOD_MS);
103        digitalWrite(led1, LOW);
104        ESP.restart();
105    }
106   // ESP_LOGI(TAG, "File open");
107    fwrite(data, sizeof(char), strlen(data), f);
108    fclose(f);
109    //ESP_LOGI(TAG, "File written");
110
111 }
112
113 void preparing_hexdump(void* arg)
114 {
115    ESP_LOGI(TAG, "Core %d for time intensive operation active!",
           xPortGetCoreID());
116    while (1) {
117        // Daten aus Ringpuffer nehmen
118        size_t len;
119        wifi_promiscuous_pkt_t* packet =
               (wifi_promiscuous_pkt_t*)xRingbufferReceive(packetRingbuf,
               &len, portMAX_DELAY);
120        if (len == 1) {
121            //Meta-packet to free up ringbuffer and exit thread.
122            vRingbufferReturnItem(packetRingbuf, packet);
123            vRingbufferDelete(packetRingbuf);
124            vTaskDelete(NULL);
125        }
126
127        //Parsen der Adressen (siehe Kapitel 2.3.2 Aufbau des Frame
               Formats)
128        uint8_t address1[12], address2[12], address3[12],
               address4[12];
129        for (int i = 0; i < 6; i++) {
130            address1[i] = { packet->payload[4 + i] };
131            address2[i] = { packet->payload[10 + i] };
132            address3[i] = { packet->payload[16 + i] };
133            address4[i] = { packet->payload[24 + i] };
134        }
```

```
135         //memcmp returned Wert 0 wenn mac_filter mit address1, -2, -3
                oder -4 bereinstimmt, andernfalls != 0
136         int a1, a2, a3, a4;
137         a1 = memcmp(mac_filter, address1, sizeof(mac_filter));
138         a2 = memcmp(mac_filter, address2, sizeof(mac_filter));
139         a3 = memcmp(mac_filter, address3, sizeof(mac_filter));
140         a4 = memcmp(mac_filter, address4, sizeof(mac_filter));
141
142         if ((a1 == 0) || (a2 == 0) || (a3 == 0) || (a4 == 0)) {
143            digitalWrite(led3, HIGH);
144            //DEAUTHENTIFICATION true: Lege Adresse 1 oder 3 in
                   client[] ab und beende Funktion an dieser Stelle
145            if (client_sniffing == 1) {
146               //Abfrage des DS Status um herauszufinden, welche
                      Adresse der Client ist
147               //(siehe Tabelle 2.2: Erlaeuterung des Distribution
                      Systems)
148               sprintf(buf, "%02X\n", packet->payload[1]);
149               char* ds_status = HexToBinary.convert(buf);
150
151               //Parsen des DS in int -> To DS bei [8], From DS bei [7]
152               int from_ds = ds_status[8] - '0';
153               int to_ds = ds_status[7] - '0';
154
155               if ((from_ds == 0 && to_ds == 0) || (from_ds == 0 &&
                      to_ds == 1)) {
156                  //Adresse 1 ist Client
157                  for (int i = 0; i < 6; i++) {
158                     client[i] = packet->payload[4 + i];
159                  }
160               }
161               else if ((from_ds == 1 && to_ds == 0) || (from_ds == 1
                      && to_ds == 1)) {
162                  //Adresse 3 ist Client
163                  for (int i = 0; i < 6; i++) {
164                     client[i] = packet->payload[16 + i];
165                     client_sniffing = 0;
166                  }
167               }
168               client_sniffing = 0;
169            }
170            else {
171               //DEAUTHENTIFICATION false: Ablage in client[]
                      ueberspringen und Hexdump aufbereiten
172               /*Hexdump*/
173               //sig_len enthaelt Laenge des Frames
174               for (int i = 0; i < packet->rx_ctrl.sig_len; i++) {
175                  if (i % 8 == 0) {
176                     //Offset hinzufuegen
177                     snprintf(hexbuf, sizeof(hexbuf), "%06X ", offset);
178                     strcat(hexbuf_dump, hexbuf);
179                     offset += 8;
```

```
180                 }
181                 //Hex anfuegen
182                 snprintf(hexbuf, sizeof(hexbuf), "%02X ",
                        packet->payload[i]);
183                 strcat(hexbuf_dump, hexbuf);
184                 //Absatz, wenn: 8x Hex oder Frame zuende
185                 if ((i % 8 == 7) || ((i + 1) ==
                        packet->rx_ctrl.sig_len )){
186                    snprintf(hexbuf, sizeof(hexbuf), "\n");
187                    strcat(hexbuf_dump, hexbuf);
188                 }
189              }
190               digitalWrite(led3, LOW);
191               offset = 0;
192               //printf("%s", hexbuf_dump);
193
194               save_to_sdcard(hexbuf_dump);
195               memset(&hexbuf_dump[0], 0, sizeof(hexbuf_dump));
196            }
197
198         }
199         vRingbufferReturnItem(packetRingbuf, packet);
200      }
201   }
202
203   void deauthentification_attack()
204   {
205      client_sniffing = 1;
206      esp_wifi_set_promiscuous(true);
207      while (client_sniffing == 1) { //zufaelligen Client suchen
208           printf("suche mac...\n");
209      }
210      esp_wifi_set_promiscuous(false);
211      char* packet = Deauth.builder(mac_filter, client);
212
213      //WiFi Driver zum versenden initialisieren
214      esp_interface_t wifi_if;
215      void* wifi_eth = NULL;
216      wifi_if = tcpip_adapter_get_esp_if(wifi_eth);
217
218      for (int i = 0; i < 200; i++) {
219         //esp_wifi_internal_tx(wifi_if, (void*)packet,
              sizeof(packet));
220         int len = sizeof(packet);
221         esp_err_t esp_wifi_80211_tx(wifi_interface_t ifx, const void
              *packet, int len);
222         ESP_LOGI(TAG, "send packet");
223         vTaskDelay(100 / portTICK_RATE_MS);
224      }
225      vTaskDelay(10000 / portTICK_RATE_MS);
226      Init_success = 1;
227   }
```

```
228
229 void Initialisierung()
230 {
231    sdmmc_host_t host = SDMMC_HOST_DEFAULT();
232    host.max_freq_khz = SDMMC_FREQ_HIGHSPEED;
233
234    // To use 1-line SD mode, uncomment the following line:
235    host.flags = SDMMC_HOST_FLAG_1BIT;
236
237    //Standard-Initialisierung beachtet nicht CD und WP Pins bei
          SD-Karte
238    //Sollten diese genutzt werden muss slot_config.gpio_cd und
          slot_config.gpio_wp modifiziert werden
239    sdmmc_slot_config_t slot_config = SDMMC_SLOT_CONFIG_DEFAULT();
240
241    esp_vfs_fat_sdmmc_mount_config_t mount_config = {
242       .format_if_mount_failed = true, //true, wenn bei Mountfehler
             automatisch SD-Karte formatiert werden soll
243       .max_files = 5
244    };
245    esp_err_t ret = esp_vfs_fat_sdmmc_mount("/sdcard", &host,
          &slot_config, &mount_config, &card); //mounten
246
247    if (ret != ESP_OK) {
248       if (ret == ESP_FAIL) {
249          ESP_LOGE(TAG, "Mounten des Dateisystems fehlgeschlagen. "
250                        "Wenn die SD-Karte automatisch formatiert werden
                            soll -> format_if_mount_failed = true.");
251          digitalWrite(led1, HIGH);
252          vTaskDelay(500 / portTICK_PERIOD_MS);
253          digitalWrite(led1, LOW);
254          ESP.restart();
255       }
256       else {
257          ESP_LOGE(TAG, "Initialisieren der SD-Karte fehlgeschlagen
                (%d). ", ret);
258          digitalWrite(led1, HIGH);
259          vTaskDelay(500 / portTICK_PERIOD_MS);
260          digitalWrite(led1, LOW);
261          ESP.restart();
262       }
263    }
264    //SD-Karte erfolgreich initialisiert. Name, Typ, Speicherplatz,
          etc im Serial Monitor ausgeben
265    sdmmc_card_print_info(stdout, card);
266
267    tcpip_adapter_init();
268    wifi_init_config_t cfg = WIFI_INIT_CONFIG_DEFAULT();
269    ESP_ERROR_CHECK(esp_wifi_init(&cfg));
270    ESP_ERROR_CHECK(esp_event_loop_init(event_handler, NULL));
271    ESP_ERROR_CHECK(esp_wifi_set_storage(WIFI_STORAGE_RAM));
272    ESP_ERROR_CHECK(esp_wifi_set_mode(WIFI_MODE_NULL));
```

```
273     ESP_ERROR_CHECK(esp_wifi_start());
274
275     esp_wifi_set_channel(channel, WIFI_SECOND_CHAN_NONE);
276
277     //Initialisierung erfolgreich, LED_READY blinkt auf
278     for (int i = 0; i < 4; i++) {
279        digitalWrite(led2, HIGH);
280        vTaskDelay(100 / portTICK_PERIOD_MS);
281        digitalWrite(led2, LOW);
282        vTaskDelay(100 / portTICK_PERIOD_MS);
283     }
284
285        xTaskCreatePinnedToCore(
286        preparing_hexdump, /* Function to implement the task */
287        "preparing_hexdump", /* Name of the task */
288        8192, /* Stack size in words */
289        NULL, /* Task input parameter */
290        1, /* Priority of the task */
291        NULL, /* Task handle. */
292        1); /* Core where the task should run */
293
294     esp_wifi_set_promiscuous_rx_cb(&wifi_promiscuous);
295
296     if (DEAUTHENTIFICATION) deauthentification_attack();
297
298     esp_wifi_set_promiscuous(true);
299     Init_success = 1;
300
301 }
302
303 void loop()
304 {
305     unsigned long currentMillis = millis();
306
307     //Abfrage ob Drucktaster betaetigt wurde -> Wenn ja, rufe
            Funktion "Initialisierung" auf
308     buttonState = digitalRead(button);
309     if (buttonState == HIGH) {
310        if (Init_success == 0) {
311           Initialisierung();
312        }
313        //Sobald Drucktaster erneut bestaetigt: SD-Karte unmounten und
314        //aus dem Loop ueber Restart ausbrechen
315        else {
316           Init_success = 0;
317           esp_vfs_fat_sdmmc_unmount(); //SD-Karte unmounten
318
319           Init_success = 0;
320           digitalWrite(led1, LOW);
321           digitalWrite(led2, HIGH);
322           vTaskDelay(400 / portTICK_PERIOD_MS);
323           digitalWrite(led2, LOW);
```

```
324             vTaskDelay(50 / portTICK_PERIOD_MS);
325             ESP.restart();
326         }
327     }
328
329     //Wake-up fuer Powerbank, LED_PAKET wird als Signal genutzt ->
            alle 5 Sekunden triggern
330     else if (Init_success == 0) {
331         if (currentMillis - previousMillis >= 5000) {
332             previousMillis = currentMillis;
333             digitalWrite(powerbank, HIGH);
334             vTaskDelay(50 / portTICK_PERIOD_MS);
335             digitalWrite(powerbank, LOW);
336         }
337     }
338
339     //Kanalwechsel, wenn dieser true, Initialisierung erfolgreich
            und Switching Time erreicht ist
340     if (CHANNEL_SWITCHING && Init_success == 1 && (currentMillis -
            previousMillis >= SWITCHING_TIME / portTICK_PERIOD_MS)) {
341         previousMillis = currentMillis;
342         channel = (channel % 13) + 1;
343         esp_wifi_set_channel(channel, WIFI_SECOND_CHAN_NONE);
344     }
345 }
346
347 esp_err_t event_handler(void* ctx, system_event_t* event)
348 {
349     return ESP_OK;
350 }
351
352 void wifi_promiscuous(void* buffer, wifi_promiscuous_pkt_type_t
        type)
353 {
354     //Buffer parsen
355     wifi_promiscuous_pkt_t* packet = (wifi_promiscuous_pkt_t*)buffer;
356
357     /*Callback aktiv + Client Sniffing aktiviert = Deauth/4 way
            handshake sniffing -> nur Datenpakete
358     oder: Callback aktiv + Client Sniffing deaktiviert = alle Pakete
            sniffen*/
359     if ((client_sniffing == 1 && type == WIFI_PKT_DATA) ||
            client_sniffing == 0) {
360         //letzte 4 Bytes bei Management-Paketen abschneiden (Bug im
                ESP-IDF)
361         //if (type == WIFI_PKT_MGMT) (packet->rx_ctrl.sig_len) -= 4;
362         xRingbufferSend(packetRingbuf, packet,
                packet->rx_ctrl.sig_len, 1);
363     }
364 }
```

Programmcode A.1: esp32_promiscuous.cpp (C++)

B.2 HexToBinary.cpp / HexToBinary.h

```
#ifndef HexToBinary_h
#define HexToBinary_h

#include <stdint.h>

class HexToBinary
{
 public:
   HexToBinary();
   char* convert(char hexdec[]);
};

#endif
```

Programmcode A.2: HexToBinary.h (C++)

```
/* HexToBinary Headerfile
   - Convert Hexdata into binary
   (C) 2017, Eric Schroeder */

#include "HexToBinary.h"
#include <stdio.h>
#include <string.h>

HexToBinary::HexToBinary() {

}

char* HexToBinary::convert(char hexdec[])
{
   size_t n = strlen(hexdec) - 1;
   char buf[256];
   char value[4];
   for (int i = 0; i < n; i++) {
   {
       switch(hexdec[i])
       {
          case '0': strncpy(value, "0000", 4); break;
          case '1': strncpy(value, "0001", 4); break;
          case '2': strncpy(value, "0010", 4); break;
          case '3': strncpy(value, "0011", 4); break;
          case '4': strncpy(value, "0100", 4); break;
          case '5': strncpy(value, "0101", 4); break;
          case '6': strncpy(value, "0110", 4); break;
          case '7': strncpy(value, "0111", 4); break;
          case '8': strncpy(value, "1000", 4); break;
          case '9': strncpy(value, "1001", 4); break;
          case 'A': strncpy(value, "1010", 4); break;
          case 'B': strncpy(value, "1011", 4); break;
          case 'C': strncpy(value, "1100", 4); break;
          case 'D': strncpy(value, "1101", 4); break;
```

```
36          case 'E': strncpy(value, "1110", 4); break;
37          case 'F': strncpy(value, "1111", 4); break;
38          case 'a': strncpy(value, "1010", 4); break;
39          case 'b': strncpy(value, "1011", 4); break;
40          case 'c': strncpy(value, "1100", 4); break;
41          case 'd': strncpy(value, "1101", 4); break;
42          case 'e': strncpy(value, "1110", 4); break;
43          case 'f': strncpy(value, "1111", 4); break;
44          default: printf("Ungueltiger Parameter");
45          }
46          sprintf(buf + strlen(buf), "%s", value);
47      }
48   }
49   //ueberspringe erste 4 Zeichen (Pointer auf 4 setzen) ->
        Fehlerhaft aufgrund buf + strlen(buf)
50   char* binary = buf + 4;
51   return binary;
52 }
```

Programmcode A.3: HexToBinary.cpp (C++)

B.3 deauth.cpp / deauth.h

```
#ifndef Deauth_h
#define Deauth_h

#include <stdint.h>

class Deauth
{
 public:
   Deauth();
   char* builder(uint8_t ap[], uint8_t client[]);
};

#endif
```

Programmcode A.4: deauth.h (C++)

```
/* Deauth Headerfile
   - Building Deauthentification Paket
   (C) 2017, Eric Schroeder */

#include "Deauth.h"
#include <stdio.h>
#include <string.h>

Deauth::Deauth() {

}
char* Deauth::builder(uint8_t ap[], uint8_t client[]){
   //packet declarations
   char buf[256];
   char deauthPacket[26] = {
      0xC0, 0x00, //type und subtype (c0 -> deauth)
      0x00, 0x00, //Duration
      0xBB, 0xBB, 0xBB, 0xBB, 0xBB, 0xBB, //Ziel (Client)
      0xCC, 0xCC, 0xCC, 0xCC, 0xCC, 0xCC, //Absender (AP)
      0xCC, 0xCC, 0xCC, 0xCC, 0xCC, 0xCC, //BSSID (AP)
      0x00, 0x00, //Fragment und Sequenz
      0x01, 0x00 //reason code (1 = unspecified reason)
   };

   //Paket aufbauen -> deauthPacket + Mac Client + Mac AP
   uint8_t packetSize = 0;
   uint8_t packetbuilding[26];
   for (int i = 0; i < sizeof(deauthPacket); i++) {
      packetbuilding[i] = { deauthPacket[i] };
      packetSize++;
   }

   for (int i = 0; i < 6; i++) {
      packetbuilding[4 + i] = client[i]; //Ziel (Client)
```

```
35          packetbuilding[10 + i] = packetbuilding[16 + i] = ap[i];
                //Absender (AP)
36      }
37
38      sprintf(buf, "%s", packetbuilding);
39      printf("%s", buf);
40      char* build = buf;
41      return build;
42  }
```

Programmcode A.5: deauth.cpp (C++)

B.4 Kconfig.projbuild

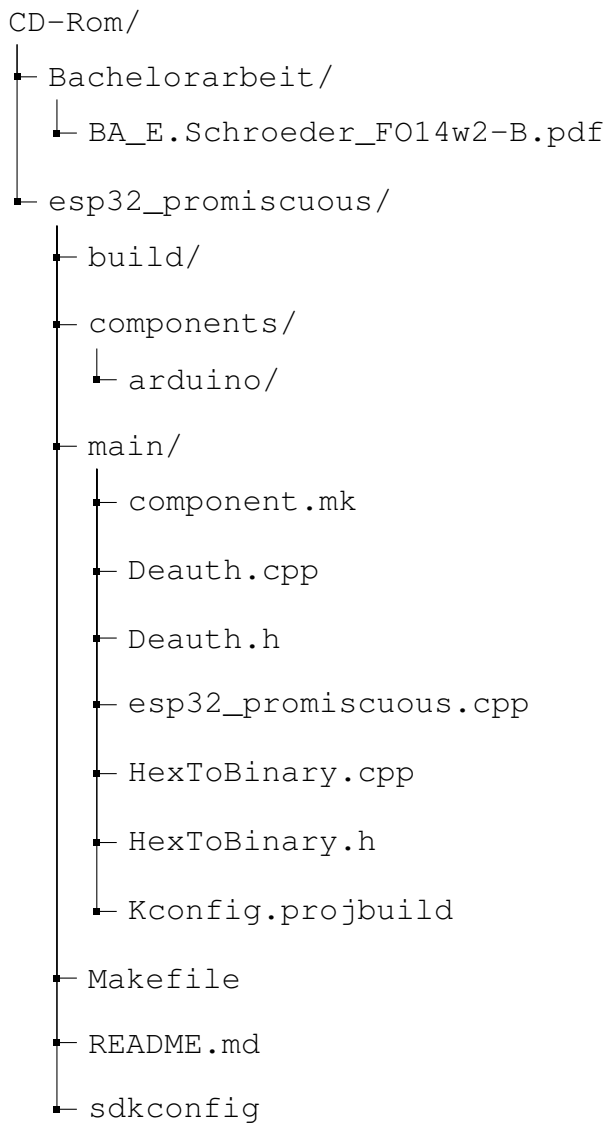
```
menu "Promiscuous Mode"

menu "WLAN Konfiguration"

config MAC_FILTER_PARA
   string "MAC-Filter"
   default "AABBCCDDEEFF"
   help
      Bestimmte MAC-Adresse sniffen. Format: AABBCCDDEEFF (ohne
         Doppelpunkte/Leerzeichen). Frei lassen um alle Access
         Points in der Umgebung zu sniffen.


config SWITCHING_TIME
   int "Zeitintervall Kanalwechsel (ms)"
   default "50"
   help
      Zeit (in ms) bis Kanal gewechselt wird


config DEAUTHENTIFICATION
   bool "BETA. Deauthentification Attack"
   default "=n"
   help
      True, wenn EAPOL Four-way-handshake benoetigt wird. Eine
         Deauthentification Attack wird zum Programmstart
         ausgefuehrt. Hinweis: Dies ist eine Betafunktion, es
         werden derzeit keine Pakete versendet.

endmenu

menu "GPIO Outputs"

config LED_ERROR
   int "GPIO fuer error LED."
   range 0 34
   default "21"
   help
      GPIO fuer error LED. Diese LED blinkt bei
         Komplikationen/Fehlermeldungen.

config LED_READY
   int "GPIO fuer ready LED"
   range 0 34
   default "22"
   help
      GPIO fuer ready LED. Diese LED blinkt bei korreketer
         Initialisierung und leuchtet auf wenn das Skript gestoppt
         wird.

```

```
43 config LED_PAKET
44    int "GPIO fuer paket LED"
45    range 0 34
46    default "19"
47    help
48       GPIO fuer paket LED. Blinkt fuer jedes empfangene Paket auf.
49
50 config BUTTON
51    int "GPIO fuer Drucktaster"
52    range 0 34
53    default "23"
54    help
55       Drucktaster um Skript zu starten/stoppen.
56
57 endmenu
58
59
60 config FILENAME
61    string "Dateiname der Capture-Datei (ohne Dateiendung)"
62    default "esp32"
63    help
64       Name der Capture-Datei (ohne Dateiendung) festlegen.
            Dateiendung wird automatisch hinzugefuegt.
65
66
67 endmenu
```

Programmcode A.6: Kconfig.projbuild (PROJBUILD-Datei)

C CD-Rom

Der Ordner „Bachelorarbeit“ enthält die digitale Fassung dieser Arbeit im .pdf-Format. Der Programmablauf ist im Ordner „esp32_promiscuous“ zu finden.



Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich meine Arbeit selbstständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und die Arbeit noch nicht anderweitig für Prüfungszwecke vorgelegt habe.

Stellen, die wörtlich oder sinngemäß aus Quellen entnommen wurden, sind als solche kenntlich gemacht.

Mittweida, 06. September 2017